A Type System For Certified Runtime Type Analysis

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Bratin Saha

Dissertation Director: Professor Zhong Shao

December 2002

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
DEC 2002		-

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
A Type System For Certified Runtime Type Analysis	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Defense Advanced Research Projects Agency,3701 North Fairfax Dr,Arlington,VA,22203-1714**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Modern programming paradigms are increasingly giving rise to applications that require type information at runtime. For example, services like garbage collection, marshalling, and serialization need to analyze type information at runtime. When compiling code which uses runtime type inspections, most existing compilers use untyped intermediate languages and discard type information at an early stage. Unfortunately, such an approach is incompatible with type-based certifying compilation. A certifying compiler generates not only the code but also a proof that the code obeys a security policy. Therefore, one need not trust the correctness of a compiler generating certified code, instead one can verify the correctness of the generated code. This allows a code consumer to accept code from untrusted sources which is specially advantageous in a networked environment. In practice, most certifying compilers use a type system for generating and encoding the proofs. These systems are called type-based certifying compilers. This dissertation describes a type system that can be used for supporting runtime type analysis in type-based certifying compilers. The type system has two novel features. First, type analysis can be applied to the type of any runtime value. In particular quantified types such as polymorphic and existential types can also be analyzed, yet type-checking remains decidable. This allows the system to be used for applications such as a copying garbage collector. Type analysis plays the key role in formalizing the contract between the mutator and the collector. Second, the system integrates runtime type analysis with the explicit representation of proofs and propositions. Essentially, it integrates an entire proof language into the type system for a compiler intermediate language. Existing certifying compilers have focussed only on simple memory and control-flow safety. This system can be used for certifying more advanced properties while still retaining decidable type-checking.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**219**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Abstract

A Type System For Certified Runtime Type Analysis

Bratin Saha

2002

Modern programming paradigms are increasingly giving rise to applications that require type information at runtime. For example, services like garbage collection, marshalling, and serialization need to analyze type information at runtime. When compiling code which uses runtime type inspections, most existing compilers use untyped intermediate languages and discard type information at an early stage. Unfortunately, such an approach is incompatible with type-based certifying compilation.

A certifying compiler generates not only the code but also a proof that the code obeys a security policy. Therefore, one need not trust the correctness of a compiler generating certified code, instead one can verify the correctness of the generated code. This allows a code consumer to accept code from untrusted sources which is specially advantageous in a networked environment. In practice, most certifying compilers use a type system for generating and encoding the proofs. These systems are called type-based certifying compilers.

This dissertation describes a type system that can be used for supporting runtime type analysis in type-based certifying compilers. The type system has two novel features.

First, type analysis can be applied to the type of any runtime value. In particular quantified types such as polymorphic and existential types can also be analyzed, yet type-checking remains decidable. This allows the system to be used for applications such as a copying garbage collector. Type analysis plays the key role in formalizing the contract between the mutator and the collector.

Second, the system integrates runtime type analysis with the explicit representation of proofs and propositions. Essentially, it integrates an entire proof language into the type system for a compiler intermediate language. Existing certifying compilers have focussed only on simple memory and control-flow safety. This system can be used for certifying more advanced properties while still retaining decidable type-checking.

Acknowledgements

First and foremost, I want to thank my advisor Professor Zhong Shao for his encouragement and support during my graduate study. He was a constant source of ideas, sound advice, and enthusiasm. He exposed me to a wealth of knowledge in programming languages, and has taught me by example the skills of a good researcher. In my first summer at Yale, he put in a huge effort in helping me get up to speed with the SML/NJ compiler. I am particularly grateful for his efforts on our "Optimal Type Lifting" paper to improve my writing. I am indebted to him for all the other things that he has taught me: giving talks, writing research proposals, and much more. This work would not have been possible without the tremendous effort that he put into guiding me. I was privileged to be his student.

I would like to thank my collaborators in the FLINT group. Valery Trifonov helped out a great deal with the work on intensional type analysis. I worked with Stefan Monnier on the application of intensional type analysis to type-safe garbage collectors. Stefan was always there to lend a hand with system administration issues. I would like to thank Chris League for many useful discussions and specially for Latex related help.

I would like to thank Carsten Schurmann, Arvind Krishnamurthy, John Peterson, and Dana Angluin in the Yale Computer Science department. All of them have helped me in various ways during my graduate study and have taken an active interest in my work. Arvind and John served on my thesis committee. Carsten helped me refine my thoughts through stimulating discussions. I would also like to thank all the people in the administrative offices at Yale – specially Judy Smith and Linda Dobb for helping me out with all the paperwork.

I want to thank Andrew Appel and Robert Harper for their help in various stages of my graduate study. Andrew served on my thesis committee and also helped me in other ways. Robert Harper shaped my thoughts through several stimulating discussions. His papers are an education in type-theory.

Finally, my deepest thanks to my parents without whom I would not be here. They have provided me with loving support throughout my childhood and adulthood, and have been a great source of inspiration. I am very grateful to my brother-in-law Prasantada, my sister Gargi, Buni and Bublu for their constant love and support. I am indebted to Atrayee for her love and encouragement. Graduating during an economic depression can be particularly depressing. She was always there to cheer me up. The later part of graduate study can get particularly stressful specially in trying to tie up all the loose ends; she was always there to lend me a hand. Without Atrayee's cheerful and constant support this work would have been impossible. I also wish to thank all my friends in Yale and outside – specially Narayan and Kishnan.

This work was sponsored in part by the Defense Advanced Research Projects Agency under contract F30602-99-1-0519 and by the National Science Foundation under contracts CCR 9633390, CCR 9901011, CCR 0081590. I am grateful to both these organizations.

Contents

Chapter 1

Introduction

This dissertation describes a type system for supporting runtime type analysis in a certifying compiler. Why is this important ? Runtime type analysis plays a crucial role in many applications, and modern language implementations are progressively targeting the generation of certified code. The following sections show how runtime type analysis plays a crucial role, and why it is important to support it in a certifying framework.

1.1 Runtime type analysis

Modern programming paradigms are increasingly giving rise to applications that rely critically on type information at runtime. For example:

- Java adopts dynamic linking as a key feature, and to ensure safe linking, an external module must be dynamically verified to satisfy the expected interface type.
- A garbage collector must keep track of all live heap objects, and for that type information must be kept at runtime to allow traversal of data structures.
- In a distributed computing environment, code and data on one machine may need to be pickled for transmission to a different machine, where the unpickler reconstructs the data structures from the bit stream. If the type of the data is not statically known at the destination (as is the case for the environment components of function closures), the unpickler must use type information, encoded in the bit stream, to correctly interpret the encoded value.

- Type-safe persistent programming requires language support for dynamic typing: the program must ensure that data read from a persistent store is of the expected type.

- Finally, in polymorphic languages like ML, the type of a value may not be known statically; therefore, compilers have traditionally used inefficient, uniformly boxed data representation. To avoid this, several modern compilers use runtime type information to support unboxed data representation.

Existing compilers can not type-check the code that involves runtime type inspections. When compiling such code they use untyped intermediate languages, and reify runtime types into values at some early stage. However, discarding the type information during compilation makes this approach incompatible with certifying compilation.

1.2 Certifying compilation

A certifying compiler [Nec98] generates not only the object code, but also a machine-checkable proof that the code satisfies a given security policy. A valid proof is an incontrovertible certificate of safety. Both the code and the proof are shipped to the consumer. Before execution, the consumer checks that the proof is correct and that it follows logically from the code.

Code certification is appealing for a number of reasons. If a compiler can generate certified code, we no longer need to trust the correctness of the compiler; instead, we can verify the correctness of the generated code. Checking a compiler-generated proof is much easier than proving the correctness of a compiler. Second, with the growth of web-based computing, programs and services are increasingly being developed or hosted at remote sites, and then downloaded by clients for execution. Client programs may also download modules dynamically as they need them. In a certifying system, clients can accept code from untrusted sources and verify it before execution.

Certified compilation is simply a general approach for applying programming language techniques for constructing secure systems. We still have to address the problem of constructing a certifying compiler. How does one arrange for a compiler to generate both the code and the proof of its safety ? Morrisett *et al.* [MWCG98] showed that a fully type-preserving compiler is a practical basis for a certifying compiler. Unlike in a conventional compiler, a type preserving compiler [Sha97b, TMC+96] does not discard types after the source program has been type-checked. Instead, each phase propagates the type information so that the code finally generated by the com-

piler can be type-checked. The safety policy enforced is type-safety, and the proof of the safety lies in the type annotations in the generated code.

Therefore, a certifying compiler for type analyzing applications must support runtime type analysis in a type-preserving framework. This dissertation describes a type system that makes this possible. The following sections describe two key features included in the type system for writing certified applications.

1.3 Fully reflexive type analysis

Many type-analyzing applications must operate on arbitrary runtime values. For example, a pickler must be able to pickle any value on the heap. Similarly, a garbage collector must be able to traverse all data structures on the heap to track live objects. Therefore, a language must support the analysis of the type of all runtime values; we call this fully reflexive type analysis.

Supporting type analysis in a type-safe framework has been an active area of research. Previous researchers in type-directed compilation [HM95, CW99] have designed frameworks for runtime type analysis. Unfortunately, none of these frameworks can support fully reflexive type analysis. Consequently, none of these approaches can be used for writing type-analyzing services like a garbage collector. Type-safe services like garbage collectors are desirable for many reasons. First, the security of a computing system depends on the safety of the underlying garbage collector. In existing systems, it is part of the trusted computing base (TCB): that part of the code that is assumed to be correct and remains unverified. Writing it in a type-safe language and verifying independently will take it out of the TCB. This makes a system more secure, specially because these services are often complex pieces of code and can introduce subtle bugs. Moreover, a type-safe implementation would make the interface to the garbage collector explicit, and type-checking could then ensure that the interface is always respected.

The type system presented in this dissertation supports fully reflexive type analysis. In particular, it supports the analysis of quantified types like polymorphic and existential types. This is the first work that shows how to support the analysis of arbitrary quantified types. We also show that the system can be used for writing a type-safe copying garbage collector. Type analysis plays the key role in capturing the contract between the mutator and the collector.

1.4 Integrating type analysis with a proof system

Existing type-based certifying compilers have focussed only on proving conventional type-safety properties (like simple memory and control-flow safety) for the generated code. On the other hand, the concept of certifying compilation, as pioneered by Necula and Lee through their proof carrying code (PCC) framework [Nec98], involves a logical system that can be used for certifying more complex specifications. For example, the foundational proof carrying code (FPCC) system [AF00b] can certify any property expressible in Church's higher-order logic. Type-based certifying compilers cannot certify properties that are as general because the type systems can not match the expressiveness of the logic used in these frameworks.

One of the challenges then in type-based certifying compilation is to come up with type systems that can be used for certifying more advanced properties than conventional type-safety. In essence, we want to express logical assertions and propositions through the type system.

The main motivation for this line of work comes from the close relationship between logical systems and type systems. The Curry-Howard isomorphism [How80] (also known as the *formula-as-types* principle) shows how proofs and propositions map into a typed lambda calculus: types represent propositions and the expressions represent proofs. For example, a proof of an implication $P \supset Q$ can be considered a function object: it is a function that when applied to a proof of proposition P yields a proof of proposition Q. Most type-based proof assistants are based on this principle. Barendregt *et al.* [Bar99, Bar91] give a good survey of previous work in this area.

Accordingly, in Chapter 5, we show a type system that integrates type analysis with the explicit representation of proofs and propositions. As far as we know, our work is the first comprehensive study on how to incorporate higher-order predicate logic and runtime type analysis into a single type system for compiler intermediate languages. Since the type system can now internalize a very expressive logic, formal reasoning traditionally done at the meta level can now be expressed inside the actual language itself.

1.5 Summary of contributions

The core contribution of this dissertation is a type system for analyzing the type of runtime values, but this system has other important ramifications. The type system can be used for type-checking the *copy* function in a stop-and-copy garbage collector, and thus provides a significant basis for

writing provably type-safe garbage collectors. The underlying ideas can also be used for integrating logical assertions in a type system, and enforcing more sophisticated invariants. To sum up:

- We show the design of a type system that supports the analysis of quantified types, both at the type level and at the term level. We prove that the resulting type system is strongly normalizing and confluent. We also show an encoding of the calculus in a type-erasure semantics. We prove that this encoding is sound by establishing a correspondence with reductions in an untyped language.

- We show that this type system can be applied for writing the *copy* function in a copying garbage collector in a type-safe manner. We show that type-checking this function relies crucially on the ability to analyze quantified types, like existential types. We prove that the language in which the *copy* function is written is sound. Our formalization does exclude some features of an industrial-strength collector, nevertheless it represents a significant step towards designing a type system that can be used for realistic garbage collectors.

- We show that the ideas (underlying the analysis of quantified types) can be extended to integrate runtime type analysis with the explicit representation of logical proofs and propositions. Again, we prove that the resulting type system is strongly normalizing and confluent, and the underlying logic is consistent.

- We show empirically that the type system can be used in an industrial-strength compiler. For this, we implemented the system in an experimental version of the SML/NJ compiler and compared its performance with version 110.34 of the compiler. On a set of large benchmarks, our measurements show that the new type system incurs a reasonable compilation overhead.

1.6 Outline of this dissertation

In Chapter 2 we give a brief overview of the typed lambda calculi. This is a useful introduction for much of the type-theoretic work presented in the later part of the dissertation. Chapters 3 through 6 constitute the core of this dissertation. In Chapter 3, we show in detail the design of the type system for fully reflexive type analysis. We prove its meta-theoretic properties in Appendix A. Chapter 4 shows how this type system can be applied for writing a type-safe copying garbage collector. The meta-theoretic properties of the languages are proved in Appendix B. In Chapter 5 we show how

the type system can be augmented to explicitly represent and manipulate proofs and propositions. Appendix C proves the meta-theoretic properties of this type system. In Chapter 6 we describe the implementation of the type system in an experimental version of the Standard ML of New Jersey compiler. We also compare the performance of our implementation to that of the SML/NJ compiler (version 110.34) on some large benchmarks.

History

The contents of Chapters 3 and Appendix A are based on [STS00b]. This work was also published as [TSS00] and [STS00a]. The contents of Chapter 4 and Appendix B are based on [MSS00]. This work was also published as [MSS01]. The contents of Chapter 5 and Appendix C are based on [SSTP01]. This work was published as [SSTP02].

Chapter 2

Overview Of The Typed Lambda Calculi

In this chapter, we will give a brief overview of the typed lambda calculi. This will ease the transition to the type systems presented later in the dissertation. The typed λ-calculi is actually an infinite sequence of languages, each more powerful than the other. The sequence starts off with the simply-typed lambda calculus and culminates in F_ω. In the last section we will also give a brief overview of the Pure Type Systems. We will show how a Pure Type System may be used for modelling many of these typed lambda calculi, and even type systems with dependent types. There is a lot of excellent literature available on these topics: the reader may refer to [PDM89, Geu93, Bar91].

2.1 The simply typed lambda calculus

The pure simply typed lambda calculus, also called F_1, may be defined as in Figure 2.1. Here τ is the set of types, e is the set of terms, x ranges over variables, $\lambda x\!:\!\tau.\, e$ is a lambda term, and $e\, e'$ is an application. In the term $\lambda x\!:\!\tau.\, e$, the variable x is the bound variable in the function body e.

$$(types) \quad \tau ::= \tau \to \tau'$$

$$(terms) \quad e ::= x \mid \lambda x\!:\!\tau.\, e \mid e\, e'$$

Figure 2.1: Syntax of F_1

$$
\begin{array}{lll}
(prog) & P & ::= De \mid DP \\
(ind) & D & ::= \mathsf{Ind}\ I\ \mathsf{with}\ (C) \\
(constr) & C & ::= x{:}\tau \mid x{:}\tau; C \\
(types) & \tau & ::= I \mid \tau \to \tau' \\
(terms) & e & ::= x \mid \lambda x{:}\tau.\, e \mid e\, e'
\end{array}
$$

Figure 2.2: Syntax of F_1^i

The reduction rule is the β-reduction where the bound variable is substituted by the argument in the function body: $(\lambda x{:}\tau.\, e)\, e' \rightsquigarrow [e'/x]e$.

Unfortunately, the language above is not very useful since the set of types is empty. To make it practical we need some mechanism of introducing primitive types and operations over these types. We could introduce some primitives in an ad-hoc manner, but we would rather use a general mechanism. This brings us to F_1^i (Figure 2.2) which is F_1 augmented with a general facility for introducing inductively defined types. Here P is a program, D is an inductive definition, C are the constructors of an inductive definition, and I is the name for an inductively defined type.

For example, we can define the natural numbers and the booleans as:

$$\mathsf{Ind}\ Nat\ \mathsf{with}\ (zero{:}Nat; succ{:}Nat \to Nat)$$
$$\mathsf{Ind}\ Bool\ \mathsf{with}\ (true{:}Bool; false{:}Bool)$$

Associated with each inductive definition is an iterator. The iterator takes a set of functions corresponding to each constructor of an inductive definition I, a value of type I, and performs structural induction over the value. For example, an iterator over the natural numbers (iter_{Nat}) would behave as follows:

$$
\begin{array}{ll}
\mathsf{iter}_{Nat}\ zero & z\ s \rightsquigarrow z \\
\mathsf{iter}_{Nat}\ (succ\, e) & z\ s \rightsquigarrow s\ (\mathsf{iter}_{Nat}\ e\ z\ s)
\end{array}
$$

The iterator can be used to implement all the ordinary primitive recursive functions over natural numbers, like addition. Note that the iterator can return a value of any type. In essence, an inductively defined type generates an infinite set of iterators, one for each return type. To ensure that the language still remains terminating, we need to constrain the definition of inductive types suitably.

Definition 2.1.1 *The set of positively occuring variables* $\mathsf{Pos}(\tau)$ *is defined as:*

$$\begin{aligned}
\mathsf{Pos}(I) &= I \\
\mathsf{Pos}(\tau \to \tau') &= \mathsf{Neg}(\tau) + \mathsf{Pos}(\tau')
\end{aligned}$$

$$\begin{aligned}
\mathsf{Neg}(I) &= \phi \\
\mathsf{Neg}(\tau \to \tau') &= \mathsf{Pos}(\tau) + \mathsf{Neg}(\tau')
\end{aligned}$$

In an inductive definition, the defined type can occur only *positively* in the type of the arguments to the constructors. Formally, an inductively defined type I is shown below. Here the x_i are the constructors. The defined type I occurs positively in the τ_{ij}.

$$\begin{aligned}
&\mathsf{Ind}\ I \quad \mathsf{with} \\
&\qquad x_1 : \tau_{11} \to \tau_{12} \to \ldots \to I; \\
&\qquad x_2 : \tau_{21} \to \tau_{22} \to \ldots \to I; \\
&\qquad \ldots \\
&\qquad x_n : \tau_{n1} \to \tau_{n2} \to \ldots \to I
\end{aligned}$$

Each such definition generates an iteration scheme $\mathsf{iter}_I{}^\tau$ with an instance of the iterator for every type τ. The iterator takes a value of type I, a function for every constructor of I, and returns a result of type τ. The type of the iterator is:

$$\begin{aligned}
\mathsf{iter}_I{}^\tau : I &\to (\overline{\tau_{11}} \to \overline{\tau_{12}} \to \ldots \to \tau) \\
&\to (\overline{\tau_{21}} \to \overline{\tau_{22}} \to \ldots \to \tau) \\
&\qquad \ldots \\
&\to (\overline{\tau_{n1}} \to \overline{\tau_{n2}} \to \ldots \to \tau) \\
&\to \tau
\end{aligned}$$

Here $\overline{\tau_{ij}}$ is equivalent to τ_{ij} with I replaced by τ. Functionally, the iterator takes a term of type I, walks over its structure, and replaces every constructor with the function corresponding to that constructor. In addition, it also recursively processes the arguments of the constructor. Expressed as a reduction rule, this is equivalent to:

$$\mathsf{iter}_I{}^\tau\ (x_i \quad e_{i1} \ldots e_{im})\ e'_1 \ldots e'_n \rightsquigarrow e'_i(\overline{e_{i1}} \ldots \overline{e_{im}})$$

where x_i is the i^{th} constructor of I, e_{ij} are its arguments, e'_i is the function corresponding to the i^{th} constructor, and $\overline{e_{ij}}$ is the result of recursively processing the argument: that is replacing every

$$(types) \quad \tau ::= \tau \to \tau' \mid \alpha \mid \forall \alpha{:}\Omega.\, \tau$$

$$(terms) \quad e ::= x \mid \lambda x{:}\tau.\, e \mid e\, e' \mid \Lambda \alpha{:}\Omega.\, e \mid e\,[\tau]$$

Figure 2.3: Syntax of F_2

subterm e of type I with $\mathsf{iter}_I{}^{\tau}\, e\, e'_1 \ldots e'_n$.

The key point about F_1^i is that all programs are *terminating* and that all reduction sequences are *confluent*.

2.2 The polymorphically typed lambda calculus

The problem with F_1^i is that there is no way to parametrize a function over the type of its arguments. For example, we can define the identity function over the natural numbers as follows:

$$id_{Nat} = \lambda x{:}Nat.\, x$$

But the identity function makes sense for values of other types as well. In F_1^i we have to repeat the definition for every instance at which we want to use it. Instead what we would like is a mechanism to abstract the type of the arguments and then instantiate the type when we actually use the function. The identity function would now look like:

$$id = \Lambda \alpha{:}\Omega.\, \lambda x{:}\alpha.\, x \qquad id_{Nat} = id\,[Nat]$$

This says that the first argument to the identity function is a type. When this function is used, the type is passed as an explicit argument. We also say that the identity function is polymorphic since it can take arguments of any type. This brings us to the polymorphically typed lambda calculus, or F_2 [Gir72, Rey74]. The syntax for F_2 is shown in Figure 2.3.

At the type level, we have a new construct ($\forall \alpha{:}\Omega.\, \tau$) to type polymorphic functions. For now, consider the Ω classifier to be a syntactic artifact. The typing rule for polymorphic functions and

type applications says:

$$\frac{\Delta, \alpha:\Omega; \Gamma \vdash e:\tau \qquad \alpha \notin \Delta}{\Delta; \Gamma \vdash \Lambda\alpha:\Omega.\, e:\forall\alpha:\Omega.\, \tau} \qquad \frac{\Delta; \Gamma \vdash e:\forall\alpha:\Omega.\, \tau}{\Delta; \Gamma \vdash e\,[\tau']:[\tau'/\alpha]\tau}$$

Here Δ is the environment that keeps track of free type variables. Γ is the value environment; it maps a variable to its type. For example, the identity function now has the type $\forall\alpha:\Omega.\, \alpha \to \alpha$.

2.2.1 Encoding F_1^i inductive definitions in F_2

If we examine the type calculus of F_2 (Figure 2.3), then again the definition does not contain any primitive types. However, unlike in F_1, we do not need to add a mechanism for introducing primitive types and operations over these types. Instead, Böhm and Berarducci [BB95] proved that we can encode all of the inductive types of F_1^i and the associated iteration operators in F_2.

As an example we will consider the encoding of the natural numbers. Suppose the inductive type Nat is given the closed type

$$Nat \equiv \forall\alpha:\Omega.\, \alpha \to (\alpha \to \alpha) \to \alpha$$

Nat takes two arguments, one of them representing zero and the other representing the successor of a natural number. Instances of the natural numbers now have the following term representation:

$$\begin{aligned}
0 &\equiv \Lambda\alpha:\Omega.\, \lambda x:\alpha.\, \lambda f:\alpha \to \alpha.\, x \\
1 &\equiv \Lambda\alpha:\Omega.\, \lambda x:\alpha.\, \lambda f:\alpha \to \alpha.\, f\, x \\
2 &\equiv \Lambda\alpha:\Omega.\, \lambda x:\alpha.\, \lambda f:\alpha \to \alpha.\, f\, (f\, x) \\
&\ldots
\end{aligned}$$

The key idea is that the term representation of the natural numbers captures the structure of the corresponding induction in F_1^i. We said in F_1^i that an iterator can return a value of any type. Since we want to capture iteration as well, we give Nat a polymorphic type. When we are iterating over natural numbers, Nat will be instantiated to the return type. In essence, the F_2 representation of a F_1^i inductive value serves as its own iterator. In F_2 the iterator over natural numbers has the

following form:

$$\mathsf{iter}_{Nat}{}^{\tau}\ n \equiv \overline{n}\,[\tau]$$

Here n is a number defined inductively in F_1^i, while $\overline{n}$ is its representation in F_2.

We know that F_2 is powerful enough to define the primitive recursive functions, but actually we can define more. In particular, we can also define Ackermann's function in F_2. However, F_2 still remains a *strongly normalizing* language: we cannot express non-terminating computations in F_2. Moreover, all reduction sequences in F_2 are *confluent*.

2.2.2 Adding inductive types to F_2

We could still go ahead and add a mechanism for adding inductively defined types to F_2. Unlike in F_1^i, the constructors can now have polymorphic types. But as in the previous case, F_2^i can be encoded in F_3, the next language in the hierarchy. In fact, Pfenning [Pfe88] proved that inductively defined types with polymorphic constructors in the n^{th} order λ-calculus can be translated into the pure $(n+1)^{th}$ order λ-calculus.

However, with the addition of polymorphic types we need to redefine the notion of positive and negative occurrences.

Definition 2.2.1 *The set of positively occuring variables* $\mathsf{Pos}(\tau)$ *is defined as:*

$$\begin{aligned}
\mathsf{Pos}(\alpha) &= \alpha \\
\mathsf{Pos}(\tau \to \tau') &= \mathsf{Neg}(\tau) + \mathsf{Pos}(\tau') \\
\mathsf{Pos}(\forall\alpha{:}\Omega.\,\tau) &= \mathsf{Pos}(\tau)
\end{aligned}$$

$$\begin{aligned}
\mathsf{Neg}(\alpha) &= \phi \\
\mathsf{Neg}(\tau \to \tau') &= \mathsf{Pos}(\tau) + \mathsf{Neg}(\tau') \\
\mathsf{Neg}(\forall\alpha{:}\Omega.\,\tau) &= \mathsf{Neg}(\tau)
\end{aligned}$$

2.3 The third order lambda calculus

Suppose we wanted to represent type constructors in our language. Let us consider the Vector type. The type Vector is in itself not the type of any object. We need to apply it to another type before we get something that is the type of an object; for example, a vector of integers. Therefore, a vector is a form of a type constructor, or a function from types to types. Programming languages

$$
\begin{array}{lll}
(\mathit{kinds}) & \kappa & ::= \Omega \mid \Omega \to \kappa \\
(\mathit{types}) & \tau & ::= \tau \to \tau' \mid \alpha \mid \forall \alpha{:}\kappa.\, \tau \mid \lambda \alpha{:}\kappa.\, \tau \mid \tau\, \tau' \\
(\mathit{terms}) & e & ::= x \mid \lambda x{:}\tau.\, e \mid e\, e' \mid \Lambda \alpha{:}\kappa.\, e \mid e\, [\tau]
\end{array}
$$

Figure 2.4: Syntax of F_3

$$
\begin{array}{lll}
(\mathit{kinds}) & \kappa & ::= \Omega \mid \kappa \to \kappa' \\
(\mathit{types}) & \tau & ::= \tau \to \tau' \mid \alpha \mid \forall \alpha{:}\kappa.\, \tau \mid \lambda \alpha{:}\kappa.\, \tau \mid \tau\, \tau' \\
(\mathit{terms}) & e & ::= x \mid \lambda x{:}\tau.\, e \mid e\, e' \mid \Lambda \alpha{:}\kappa.\, e \mid e\, [\tau]
\end{array}
$$

Figure 2.5: Syntax of F_ω

have many examples of type constructors, for example Array and List. We can define a particular instance of the List constructor in F_2; for example, we could define a list of integers. But if we then needed a list of booleans, we would have to define a new type. Instead, what we would like is to define a List type constructor, and then create different instances from this constructor.

This brings us to F_3 where we add the ability of express functions from types to types. The syntax is shown in Figure 2.4. We denote type functions with a similar syntax to the term functions. We use $\tau\, \tau'$ to denote the application of a type τ to another type τ'. However, since we introduced functions at the type level, we need some mechanism for ensuring that our types are well-formed. For this we add a new layer that we call kinds. In essence, kinds are the "types" of types. We have a constant kind that we call Ω. This is the kind of the types of terms, for example the types $\tau \to \tau'$ and $\forall \alpha{:}\kappa.\, \tau$ belong to Ω. A type function ($\lambda \alpha{:}\Omega.\, \tau$) has the function kind ($\Omega \to \kappa$).

As before, F_3 is also a *strongly normalizing* language with all reductions being *confluent*.

2.4 The ω order lambda calculus

From F_3, we get to the languages F_4, F_5, *etc* by making the set of legal kinds larger. In other words, while F_3 allows only first order type constructors, these languages allow higher order type constructors. In the limit we reach F_ω where the kinds are completely general. The syntax of F_ω is shown in Figure 2.5.

F_ω has a number of interesting properties:

- All expressions (terms, types, and kinds) in F_ω have a unique normal form.
- ${F_\omega}^i = F_\omega$. Inductive type definitions whose constructors have types in F_ω can be translated into closed types in F_ω.
- F_n can express all functions whose totality is provable in n^{th} order arithmetic [Gir88].

2.5 Pure type systems

An alternative method of describing the typed lambda calculi is through the framework of Pure Type Systems (PTS). This is a general framework that can capture a large class of typed lambda calculi. In this section, we give a short tutorial on PTS using a very small expression language. The reader may refer to [Bar91] for details. The syntax of PTS pseudo-terms is

$$(\textit{pseudo-terms}) \quad A, B ::= c \mid X \mid \lambda X\!:\!A.\, B \mid A\ B \mid \Pi X\!:\!A.\, B$$

and the semantics is based on the usual notion of β reduction:

$$(\lambda X\!:\!A.\, B)\ A' \rightsquigarrow_\beta [A'/X]B$$

The first four productions in the syntax are familiar—they are constants c drawn from a set $\mathcal{C}$, variables, abstractions, and applications. The language is explicitly typed: The abstraction has a type annotation for the bound variable. Note that this type annotation is again an expression, in other words, types and terms share the same syntax. Moreover, the same productions form abstraction and application for both terms and types. For example, consider the following expression in F_2

$$\Lambda\alpha\!:\!\Omega.\, \lambda x\!:\!\alpha.\, \mathsf{id}[\alpha]\ x$$

In the above expression, Ω is a kind, the Λ binds a type variable α of kind Ω, the λ binds a term variable x of type α. In the body of the function, the polymorphic identity function is first applied to the type variable α and then to the term variable x. In contrast, in our PTS, the above expression would be written as

$$\lambda X\!:\!\Omega.\, \lambda X'\!:\!X.\, (\mathsf{id}\ X)\ X'$$

The Π production is a key feature and subsumes both function and polymorphic types. In fact, it is the only type forming operator in the language. Intuitively the term $\Pi X : A.\, B$ is the type of functions from values of type A to values of type B. Moreover, the type B may *depend* on the value of the argument. It is obvious that Π subsumes the function type

$$A \to B = \Pi\_ : A.\, B$$

Now consider the expression $\Pi X : \Omega.\, A$. This is the type of functions from values of type Ω to values of type A where X may occur free in A. But this is precisely the meaning of the type $\forall \alpha : \Omega.\, A'$ in F_2 (with $A' = [\alpha / X]A$).

We use a type system to define the well formed expressions—in turn, the type system relies on a specification.

Definition 2.5.1 *The* specification *of a PTS is a triple ($\mathcal{S}$, $\mathcal{A}$, $\mathcal{R}$) where*

- $\mathcal{S} \subseteq \mathcal{C}$ *is called the set of* sorts
- $\mathcal{A} \subseteq \mathcal{C} \times \mathcal{S}$ *is a set of* axioms
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ *is a set of* rules.

We will use (s_1, s_2) as an abbreviation for elements of $\mathcal{R}$ of the form (s_1, s_2, s_2). In the rest of this section we will only consider these simpler PTSs: these are sufficient to describe the type systems of interest to us. We will also use s to denote any particular sort. The formation rules for terms of the PTS ($\mathcal{S}$, $\mathcal{A}$, $\mathcal{R}$) are as follows.

$$\frac{(c, s) \in \mathcal{A}}{\vdash c : s} \qquad (\text{AX})$$

$$\frac{\Delta \vdash A : s}{\Delta, X : A \vdash X : A} \qquad (\text{VAR})$$

$$\frac{\Delta \vdash A : B \qquad \Delta \vdash C : s}{\Delta, X : C \vdash A : B} \qquad (\text{WEAK})$$

$$\frac{\Delta \vdash A : \Pi X : B'.\, A' \quad \Delta \vdash B : B'}{\Delta \vdash A\ B : [B/X]A'} \qquad (\text{APP})$$

$$\frac{\Delta, X : A \vdash B : B' \qquad \Delta \vdash \Pi X : A.\, B' : s}{\Delta \vdash \lambda X : A.\, B : \Pi X : A.\, B'} \qquad (\text{FUN})$$

$$\frac{\Delta \vdash A : s_1 \quad \Delta, X{:}A \vdash B : s_2 \quad (s_1, s_2) \in \mathcal{R}}{\Delta \vdash \Pi X{:}A.\, B : s_2} \qquad \text{(PROD)}$$

$$\frac{\Delta \vdash A : B \qquad \Delta \vdash B' : s \qquad B =_\beta B'}{\Delta \vdash A : B'} \qquad \text{(CONV)}$$

The first three rules are fairly straightforward. The AX rule merely specifies the sorts and the relation between them. In the VAR and the WEAK rules, we ensure that the type of the variable is well formed. The APP rule is more interesting. Note that in the type for A, the variable X can occur free in the body A'; therefore, the result type is obtained by substituting for the variable X. In the special case of an arrow type, the variable X does not occur free in the body A'. Therefore, the substitution has no effect and the typing derivation reduces to the usual rule for function application. Moreover, the rule is sufficient for dealing with type applications. Consider the rule for type application in F_2 that we saw before:

$$\frac{\Delta; \Gamma \vdash e : \forall \alpha{:}\Omega.\, \tau}{\Delta; \Gamma \vdash e\,[\tau'] : [\tau'/\alpha]\tau}$$

The APP rule is equivalent to the above rule when we consider that the polymorphic type $\forall \alpha{:}\Omega.\, \tau$ can be written as a product type $\Pi X{:}\Omega.\, A$, and there is only a single syntactic category of terms.

The FUN and the CONV rule hold no new surprises. The equality in the CONV rule is defined as the reflexive, symmetric and transitive closure of the one step reduction ($\leadsto_\beta$). The PROD rule describes when a Π expression is a valid type. The PTS contains as many copies of this rule as there are members in the set $\mathcal{R}$. Through this rule, the set $\mathcal{R}$ specializes the PTS to a particular type system.

Suppose we specialize a PTS to the following specification:

$$\mathcal{S} = (\Omega, \mathsf{Kind}) \qquad \mathcal{A} = (\Omega, \mathsf{Kind}) \qquad \mathcal{R} = \{(\Omega, \Omega)\}$$

In the PROD rule, both A and B must belong to Ω which means that they are both types. This implies (from the FUN rule) that we can abstract a term variable over a term expression. Since this is the only kind of abstractions that we are allowed to construct, essentially we have the simply typed lambda calculus.

Consider the PTS with the following specification:

$$\mathcal{S} = (\Omega, \mathsf{Kind}) \qquad \mathcal{A} = (\Omega, \mathsf{Kind}) \qquad \mathcal{R} = \{(\Omega, \Omega), (\mathsf{Kind}, \Omega)\}$$

We will now have two instances of the PROD rule. Specializing to (Kind, Ω) we get

$$\frac{\Delta \vdash A : \mathsf{Kind} \quad \Delta, X : A \vdash B : \Omega}{\Delta \vdash \Pi X : A.\, B : \Omega}$$

This means that A is a kind and B is a type; which implies that X is a type variable. In other words, we now have a polymorphic type. Correspondingly, we now also have polymorphic abstractions in addition to term abstractions. Therefore, the above specification describes the system F_2.

Before closing this section, we would like to show how a PTS can also specify systems with dependent types. Consider adding the pair (Ω, Kind) to $\mathcal{R}$.

$$\frac{\Delta \vdash A : \Omega \quad \Delta, X : A \vdash B' : \mathsf{Kind}}{\Delta \vdash \Pi X : A.\, B' : \mathsf{Kind}}$$

$$\frac{\Delta, X : A \vdash B : B' \qquad \Delta \vdash \Pi X : A.\, B' : \mathsf{Kind}}{\Delta \vdash \lambda X : A.\, B : \Pi X : A.\, B'}$$

In the PROD rule, B' is a kind while A is a type, and X is a term variable. In the FUN rule, since $\Pi X : A.\, B'$ is a kind, the abstraction is a function at the type level. But this function expects a term level expression as an argument. Hence, the value of this type expression depends on a term expression which implies that we are in the realm of dependent types.

The following table lists some type systems that are instances of a PTS with the above $\mathcal{S}$ and $\mathcal{A}$. They form part of what is popularly called the λ-cube.

System	$\mathcal{R}$
simply typed λ-calculus	(Ω, Ω)
F_2	$(\Omega, \Omega), (\mathsf{Kind}, \Omega)$
F_ω	$(\Omega, \Omega), (\mathsf{Kind}, \Omega), (\mathsf{Kind}, \mathsf{Kind})$
Calculus of constructions	$(\Omega, \Omega), (\mathsf{Kind}, \Omega), (\mathsf{Kind}, \mathsf{Kind}), (\Omega, \mathsf{Kind})$

Chapter 3

A Language For Runtime Type Analysis

3.1 Introduction

Runtime type analysis is used extensively in various applications and programming situations. Runtime services such as garbage collection and dynamic linking, applications such as marshalling and pickling, type-safe persistent programming, and unboxing implementations of polymorphic languages all analyze types to various degrees at runtime. Most existing compilers use untyped intermediate languages for compilation; therefore, they support runtime type inspection in a type-unsafe manner.

This chapter presents a statically typed intermediate language that allows runtime type analysis to be coded within the language. Therefore, it can be used to support runtime type analysis in a type-safe manner. The system presented here builds on existing work [HM95] but makes the following new contributions:

- It supports fully reflexive type analysis at the term level. Consequently, programs can analyze any runtime value such as function closures and polymorphic data structures.
- It supports fully reflexive type analysis at the type level. Therefore, type transformations operating on arbitrary types can be encoded in our language.
- We prove that the language is sound and that type reduction is strongly normalizing and confluent.
- We show a translation into a type erasure semantics.

By fully reflexive we mean that type analyzing operations are applicable to the type of any runtime value in the language. In particular, the language provides both type-level and term-level constructs for analyzing quantified types such as polymorphic and existential types.

3.2 Background

Harper and Morrisett [HM95] proposed intensional type analysis and presented a type-theoretic framework for expressing computations that analyze types at runtime. They introduced two explicit type-analysis operators: one at the term level (typecase) and another at the type level (Typerec); both use induction over the structure of types. Type-dependent primitive functions use these operators to analyze types and select the appropriate code. For example, a polymorphic subscript function for arrays might be written as the following pseudo-code:

$$\begin{aligned}\mathsf{sub} = \Lambda\alpha : \Omega.\, &\mathsf{typecase}\ \alpha\ \mathsf{of}\\ &\mathsf{int} \Rightarrow \mathsf{intsub}\\ &\mathsf{real} \Rightarrow \mathsf{realsub}\\ &\beta \Rightarrow \mathsf{boxedsub}\,[\beta]\end{aligned}$$

Here sub analyzes the type α of the array elements and returns the appropriate subscript function. We assume that arrays of type int and real have specialized representations (defined by types, say, intarray and realarray), and therefore special subscript functions, while all other arrays use the default boxed representation.

Typing this subscript function is more interesting, because it must have all of the types $\mathsf{intarray} \to \mathsf{int} \to \mathsf{int}$, $\mathsf{realarray} \to \mathsf{int} \to \mathsf{real}$, and $\forall\alpha : \Omega.\, \mathsf{boxedarray}\,(\alpha) \to \mathsf{int} \to \alpha$.
To assign a type to the subscript function, we need a construct at the type level that parallels the typecase analysis at the term level. In general, this facility is crucial since many type-analyzing operations like flattening and marshalling transform types in a non-uniform way. The subscript operation would then be typed as

$$
\begin{array}{lll}
(kinds) & \kappa & ::= \Omega \mid \kappa \to \kappa' \\
(cons) & \tau & ::= \mathsf{int} \mid \tau \to \tau' \mid \alpha \mid \lambda\alpha{:}\kappa.\,\tau \mid \tau\,\tau' \\
 & & \quad \mid \mathsf{Typerec}\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to}) \\
(types) & \sigma & ::= \tau \mid \forall\alpha{:}\kappa.\,\sigma
\end{array}
$$

Figure 3.1: The type language of Harper and Morrisett

$$
\begin{array}{rrcl}
 & \mathsf{sub} & : & \forall\alpha:\Omega.\,\mathsf{Array}\,(\alpha) \to \mathsf{int} \to \alpha \\
\text{where} & \mathsf{Array} & = & \lambda\alpha:\Omega.\,\mathsf{Typecase}\ \alpha\ \mathsf{of} \\
 & & & \quad \mathsf{int} \Rightarrow \mathsf{intarray} \\
 & & & \quad \mathsf{real} \Rightarrow \mathsf{realarray} \\
 & & & \quad \beta \Rightarrow \mathsf{boxedarray}\,\beta
\end{array}
$$

The Typecase construct in the above example is a special case of the Typerec construct in [HM95], which also supports primitive recursion over types.

3.2.1 The problem

The language of Harper and Morrisett splits the type language into two universes, constructors and types (Figure 3.1), with the constructors containing just the monotypes. Type analysis is restricted to the constructors; they do not support analysis of types with binding structure (*e.g.*, polymorphic, or existential types). Therefore, type analyzing primitives that handle polymorphic code blocks, or closures (since closures are represented as existentials [MMH96]), cannot be written in their language. The constructor-kind calculus (Figure 3.1) is essentially a simply typed lambda calculus augmented with the constant Ω. The Typerec operator analyzes only constructors of kind Ω:

$$
\begin{array}{rcl}
\mathsf{int} & : & \Omega \\
\to & : & \Omega \to \Omega \to \Omega
\end{array}
$$

The kinds of these constructors' arguments do not contain any negative occurence (Section 2.1) of the kind Ω, so int and $\to$ can be used to define Ω inductively. The Typerec operator is essentially

an iterator over this inductive definition; its reduction rules can be written as:

$$\mathsf{Typerec}\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow}) \rightsquigarrow \tau_{\mathsf{int}}$$

$$\begin{aligned}&\mathsf{Typerec}\ (\tau_1 \rightarrow \tau_2)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow}) \rightsquigarrow \\ &\quad \tau_{\rightarrow}\,\tau_1\,\tau_2\,(\mathsf{Typerec}\ \tau_1\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow}))\,(\mathsf{Typerec}\ \tau_2\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow}))\end{aligned}$$

Here the Typerec operator examines the head constructor of the type being analyzed and chooses a branch accordingly. If the type is int, it reduces to the τ_{int} branch. If the type is $\tau_1 \rightarrow \tau_2$, the analysis proceeds recursively on the subtypes τ_1 and τ_2. The Typerec operator then applies the $\tau_{\rightarrow}$ branch to the original component types, and to the result of analyzing the components; thus providing a form of primitive recursion.

Types with binding structure can be constructed using higher-order abstract syntax. For example, the polymorphic type constructor $\forall$ can be given the kind $(\Omega \rightarrow \Omega) \rightarrow \Omega$, so that the type $\forall\alpha:\Omega.\,\alpha \rightarrow \alpha$ is represented as $\forall\,(\lambda\alpha:\Omega.\,\alpha \rightarrow \alpha)$. It would seem plausible to define an iterator with the reduction rule:

$$\begin{aligned}&\mathsf{Typerec}\ (\forall\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall}) \\ &\rightsquigarrow\ \tau_{\forall}\,\tau\,(\lambda\alpha:\Omega.\,\mathsf{Typerec}\ \tau\,\alpha\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall}))\end{aligned}$$

However the negative occurence of Ω in the kind of the argument of $\forall$ poses a problem: this iterator may fail to terminate! Consider the following example, assuming $\tau = \lambda\alpha:\Omega.\,\alpha$ and

$$\tau_{\forall} = \lambda\beta_1:\Omega \rightarrow \Omega.\,\lambda\beta_2:\Omega \rightarrow \Omega.\,\beta_2\,(\forall\beta_1)$$

the following reduction sequence will go on indefinitely:

$$\begin{aligned}&\mathsf{Typerec}\ (\forall\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall}) \\ &\rightsquigarrow\ \tau_{\forall}\,\tau\,(\lambda\alpha:\Omega.\,\mathsf{Typerec}\ \tau\,\alpha\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall})) \\ &\rightsquigarrow\ \mathsf{Typerec}\ (\tau\,(\forall\,\tau))\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall}) \\ &\rightsquigarrow\ \mathsf{Typerec}\ (\forall\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall}) \\ &\rightsquigarrow\ \ldots\end{aligned}$$

Clearly this makes typechecking Typerec undecidable.

Another serious problem in analyzing quantified types involves both the type-level and the

term-level operators. Typed intermediate languages like FLINT [Sha97b] and TIL [Tar96] are based on the F_ω calculus [Gir72, Rey74], which has higher order type constructors. In a quantified type, say $\exists\beta\!:\!\kappa.\,\tau$, the quantified variable β is no longer restricted to the base kind Ω, but can have an arbitrary kind κ. Consider the term-level typecase in such a scenario:

$$\begin{aligned}\mathsf{sub} = \Lambda\alpha : \Omega.\, &\mathsf{typecase}\ \alpha\ \mathsf{of}\\ &\mathsf{int} \qquad \Rightarrow e_{\mathsf{int}}\\ &\ldots\\ &\exists\beta\!:\!\kappa.\,\tau \Rightarrow e_\exists\end{aligned}$$

To do anything useful in the $e_\exists$ branch, even to open a package of this type, we need to know the kind κ. We can get around this by having an infinite number of branches in the typecase, one for each kind; or by restricting type analysis to a finite set of kinds. Both of these approaches are clearly impractical. Recent work on typed compilation of ML and Java has shown that both would require an F_ω-like calculus with arbitrarily complex kinds [Sha98, Sha99, LST99].

3.2.2 Requirements for a solution

Before getting to the solution, we will enumerate the properties that it should have.

First, our language must support type analysis in the manner of Harper/Morrisett. That is, we want to include type analysis primitives that will analyze the entire syntax tree representing a type. Second, we want the analysis to continue inside the body of a quantified type; handling quantified types parametrically, or in a uniform way by providing a default case, is insufficient. As we will see later, many interesting type-directed operations require these two properties. Third, we do not want to restrict the kind of the (quantified) type variable in a quantified type; we want to analyze types where the quantification is over a variable of arbitrary kind.

Consider a type-directed pickler that converts a value of arbitrary type into an external representation. Suppose we want to pickle a closure. With a type-preserving compiler, the type of a closure would be represented as an existential with the environment held abstract. Even if the code is handled uniformly, the function must inspect the type of the environment (which is also the witness type of the existential package) to pickle it. This shows that at the term level, the analysis must proceed inside a quantified type. In Section 3.3.1, we show the encoding of a polymorphic equality function in our calculus; the comparison of existential values requires a similar technique.

The reason for not restricting the quantified type variable to a finite set of kinds is twofold. Restricting type analysis to a finite number of kinds would be *ad hoc* and there is no way of satisfactorily predetermining this finite set (this is even more the case when we compile Java into a typed intermediate language [LST99]). More importantly, if the kind of the bound variable is a known constant in the corresponding branch of the Typerec construct, then it is easy to generalize the non-termination example of the previous section and break the decidability of the type system.

3.2.3 The solution

The key problem in analyzing quantified types such as the polymorphic type $\forall\alpha:\Omega.\,\alpha\to\alpha$ is to determine what happens when the iteration reaches the quantified type variable α, or (in the general case of type variables of higher kinds) a normal form which is an application with a type variable in the head.

One approach would be to leave the type variable untouched while analyzing the body of the quantified type. The equational theory of the type language then includes a reduction of the form $(\mathsf{Typerec}\ \alpha\ \mathsf{of}\ \ldots)\rightsquigarrow\alpha$ so that the iterator vanishes when it reaches a type variable. However this would break the confluence of the type language—the application of $\lambda\alpha:\Omega.\,\mathsf{Typerec}\ \alpha\ \mathsf{of}\ldots$ to τ would reduce in general to different types if we perform the β-reduction step first or eliminate the iterator first.

Crary and Weirich [CW99] propose another method for solving this problem. Their language LX allows the representation of terms with bound variables using deBruijn notation and an encoding of natural numbers as types. To analyze quantified types, the iterator carries an environment mapping indices to types; when the iterator reaches a type variable, it returns the corresponding type from the environment. This method has several disadvantages.

- It is not fully reflexive, since it does not allow analysis of all quantified types—their analysis is restricted to types with quantification only over variables of kind Ω.
- The technique is "limited to *parametrically* polymorphic functions, and cannot account for functions that perform intensional type analysis" [CW99, Section 4.1]. For example polymorphic types such as $\forall\alpha:\Omega.\,\mathsf{Typerec}\ \alpha\ \mathsf{of}\ \ldots$ are not analyzable in their framework.
- The correctness of the structure of a type encoded using deBruijn notation cannot be verified by the kind language (indices not corresponding to bound variables go undetected, so the

environment must provide a default type for them), which does not break the type soundness but opens the door for programmer mistakes.

To account for non-parametrically polymorphic functions, we must analyze the quantified type variable. Moreover, we want to have confluence of the type language, so β-reduction should be transparent to the iterator. This is possible only if the analysis gets suspended when it reaches a type variable (or an application with a type variable in the head), and resumes when the variable gets substituted. Therefore, we consider $(\mathsf{Typerec}\ \alpha\ \mathsf{of}\ \ldots)$ to be a normal form. For example, the result of analyzing the body $(\alpha \to \mathsf{int})$ of the polymorphic type $\forall\alpha\!:\!\kappa.\,\alpha \to \mathsf{int}$ is

$$\mathsf{Typerec}\ (\alpha \to \mathsf{int})\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall}) \ \leadsto\ \tau_{\to}\,\alpha\,\mathsf{int}\,(\mathsf{Typerec}\ \alpha\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall}))\,(\tau_{\mathsf{int}})$$

We will formalize the analysis of quantified types while presenting the type reduction rules of the Typerec construct (Figure 3.6).

The other problem is to analyze quantified types when the quantified variable can be of an arbitrary kind. In our language the solution is similar at both the type and the term levels: we use kind polymorphism! We introduce kind abstractions at the type level ($\Lambda j.\,\tau$) and at the term level ($\Lambda^{+} j.\,e$) to bind the kind of the quantified variable. (See Section 3.3 for details.)

It is important to note that our language provides no facilities for kind analysis. Thus every type function of polymorphic kind is parametrically polymorphic. Analyzing the kind κ of the bound variable α in the type $\forall\,(\lambda\alpha\!:\!\kappa.\,\tau)$ would let us synthesize a type of the same kind, for every kind κ. This type could then be used to create non-terminating reduction sequences [HM99].

3.3 Analyzing quantified types

In the impredicative F_ω calculus, the polymorphic types $\forall\alpha\!:\!\kappa.\,\tau$ can be viewed as generated by an infinite set of type constructors $\forall_\kappa$ of kind $(\kappa \to \Omega) \to \Omega$, one for each kind κ. The type $\forall\alpha\!:\!\kappa.\,\tau$ is then represented as $\forall_\kappa\,(\lambda\alpha\!:\!\kappa.\,\tau)$. The kinds of constructors that can generate types of kind Ω

$$
\begin{array}{lll}
(\mathit{kinds}) & \kappa & ::= \Omega \mid \kappa \to \kappa' \mid j \mid \forall j.\,\kappa \\
\\
(\mathit{types}) & \tau & ::= \mathsf{int} \mid \twoheadrightarrow \mid \mathsf{\forall} \mid \mathsf{\forall}^{+} \\
 & & \quad \mid \alpha \mid \Lambda j.\,\tau \mid \lambda\alpha{:}\kappa.\,\tau \mid \tau\,[\kappa] \mid \tau\,\tau' \\
 & & \quad \mid \mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^{+}}) \\
\\
(\mathit{values}) & v & ::= i \mid \Lambda^{+} j.\,e \mid \Lambda\alpha{:}\kappa.\,e \mid \lambda x{:}\tau.\,e \mid \mathsf{fix}\,x{:}\tau.\,v \\
\\
(\mathit{terms}) & e & ::= v \mid x \mid e\,[\kappa]^{+} \mid e\,[\tau] \mid e\,e' \\
 & & \quad \mid \mathsf{typecase}[\tau]\ \tau'\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^{+}})
\end{array}
$$

Figure 3.2: Syntax of the λ_i^ω language

$$
\begin{aligned}
\tau \to \tau' &\equiv ((\twoheadrightarrow)\,\tau)\,\tau' \\
\forall\alpha{:}\kappa.\,\tau &\equiv (\mathsf{\forall}\,[\kappa])\,(\lambda\alpha{:}\kappa.\,\tau) \\
\forall^{+} j.\,\tau &\equiv \mathsf{\forall}^{+}(\Lambda j.\,\tau)
\end{aligned}
$$

Figure 3.3: Syntactic sugar for λ_i^ω types

then would be

$$
\begin{array}{lcl}
\mathsf{int} & : & \Omega \\
\twoheadrightarrow & : & \Omega \to \Omega \to \Omega \\
\forall_\Omega & : & (\Omega \to \Omega) \to \Omega \\
\ldots & & \\
\forall_\kappa & : & (\kappa \to \Omega) \to \Omega \\
\ldots & &
\end{array}
$$

We can avoid the infinite number of $\forall_\kappa$ constructors by defining a single constructor $\mathsf{\forall}$ of polymorphic kind $\forall j.\,(j \to \Omega) \to \Omega$ and then instantiating it to a specific kind before forming polymorphic types. More importantly, this technique also removes the negative occurrence of Ω from the kind of the argument of the constructor $\forall_\Omega$. Hence we extend F_ω with polymorphic kinds and add a type constant $\mathsf{\forall}$ of kind $\forall j.\,(j \to \Omega) \to \Omega$ to the type language. The polymorphic type $\forall\alpha{:}\kappa.\,\tau$ is now represented as $\mathsf{\forall}\,[\kappa]\,(\lambda\alpha{:}\kappa.\,\tau)$.

We define the syntax of the λ_i^ω calculus in Figure 3.2, and some derived forms of types in Figure 3.3. The static semantics of λ_i^ω is shown in Figures 3.4, 3.5 and 3.6 as a set of rules for

Kind formation $\quad \mathcal{E} \vdash \kappa$

$$\mathcal{E} \vdash \Omega \qquad \frac{j \in \mathcal{E}}{\mathcal{E} \vdash j} \qquad \frac{\mathcal{E} \vdash \kappa \qquad \mathcal{E} \vdash \kappa'}{\mathcal{E} \vdash \kappa \to \kappa'} \qquad \frac{\mathcal{E}, j \vdash \kappa}{\mathcal{E} \vdash \forall j.\, \kappa}$$

Type environment formation $\quad \mathcal{E} \vdash \Delta$

$$\mathcal{E} \vdash \varepsilon \qquad \frac{\mathcal{E} \vdash \Delta \qquad \mathcal{E} \vdash \kappa}{\mathcal{E} \vdash \Delta, \alpha : \kappa}$$

Type formation $\quad \mathcal{E}; \Delta \vdash \tau \,:\, \kappa$

$$\frac{\mathcal{E} \vdash \Delta}{\begin{array}{l}\mathcal{E}; \Delta \vdash \mathsf{int} \;:\; \Omega \\ \mathcal{E}; \Delta \vdash (\twoheadrightarrow) \,:\, \Omega \to \Omega \to \Omega \\ \mathcal{E}; \Delta \vdash \forall \;:\; \forall j.\, (j \to \Omega) \to \Omega \\ \mathcal{E}; \Delta \vdash \forall^{+} \;:\; (\forall j.\, \Omega) \to \Omega\end{array}} \qquad \frac{\mathcal{E} \vdash \Delta \qquad \alpha : \kappa \text{ in } \Delta}{\mathcal{E}; \Delta \vdash \alpha \,:\, \kappa}$$

$$\frac{\mathcal{E}, j; \Delta \vdash \tau \,:\, \kappa}{\mathcal{E}; \Delta \vdash \Lambda j.\, \tau \,:\, \forall j.\, \kappa} \qquad \frac{\mathcal{E}; \Delta \vdash \tau \,:\, \forall j.\, \kappa \qquad \mathcal{E} \vdash \kappa'}{\mathcal{E}; \Delta \vdash \tau\, [\kappa'] \,:\, [\kappa'/j]\kappa}$$

$$\frac{\mathcal{E}; \Delta, \alpha : \kappa \vdash \tau \,:\, \kappa'}{\mathcal{E}; \Delta \vdash \lambda \alpha : \kappa.\, \tau \,:\, \kappa \to \kappa'} \qquad \frac{\mathcal{E}; \Delta \vdash \tau \,:\, \kappa' \to \kappa \qquad \mathcal{E}; \Delta \vdash \tau' \,:\, \kappa'}{\mathcal{E}; \Delta \vdash \tau\, \tau' \,:\, \kappa}$$

$$\frac{\begin{array}{l}\mathcal{E}; \Delta \vdash \tau \;:\; \Omega \\ \mathcal{E}; \Delta \vdash \tau_{\mathsf{int}} \,:\, \kappa \\ \mathcal{E}; \Delta \vdash \tau_{\to} \,:\, \Omega \to \Omega \to \kappa \to \kappa \to \kappa \\ \mathcal{E}; \Delta \vdash \tau_{\forall} \;:\; \forall j.\, (j \to \Omega) \to (j \to \kappa) \to \kappa \\ \mathcal{E}; \Delta \vdash \tau_{\forall^{+}} \,:\, (\forall j.\, \Omega) \to (\forall j.\, \kappa) \to \kappa\end{array}}{\mathcal{E}; \Delta \vdash \mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^{+}}) \,:\, \kappa}$$

Figure 3.4: Type formation rules of λ_i^{ω}

Term environment formation $\mathcal{E}; \Delta \vdash \Gamma$

$$\frac{\mathcal{E} \vdash \Delta}{\mathcal{E}; \Delta \vdash \varepsilon} \qquad \frac{\mathcal{E}; \Delta \vdash \Gamma \qquad \mathcal{E}; \Delta \vdash \tau \,:\, \Omega}{\mathcal{E}; \Delta \vdash \Gamma, x{:}\tau}$$

Term formation $\mathcal{E}; \Delta; \Gamma \vdash e \,:\, \tau$

$$\frac{\mathcal{E}; \Delta; \Gamma \vdash e \,:\, \tau \qquad \mathcal{E}; \Delta \vdash \tau \rightsquigarrow \tau' \,:\, \Omega}{\mathcal{E}; \Delta; \Gamma \vdash e \,:\, \tau'} \qquad \frac{\mathcal{E}; \Delta \vdash \Gamma}{\mathcal{E}; \Delta; \Gamma \vdash i \,:\, \mathsf{int}}$$

$$\frac{\mathcal{E}; \Delta \vdash \Gamma \qquad x{:}\tau \text{ in } \Gamma}{\mathcal{E}; \Delta; \Gamma \vdash x \,:\, \tau} \qquad \frac{\mathcal{E}, j; \Delta; \Gamma \vdash v \,:\, \tau}{\mathcal{E}; \Delta; \Gamma \vdash \Lambda^{+} j.\, v \,:\, \forall^{+} j.\, \tau}$$

$$\frac{\mathcal{E}; \Delta, \alpha{:}\kappa; \Gamma \vdash v \,:\, \tau}{\mathcal{E}; \Delta; \Gamma \vdash \Lambda \alpha{:}\kappa.\, v \,:\, \forall \alpha{:}\kappa.\, \tau} \qquad \frac{\mathcal{E}; \Delta; \Gamma, x{:}\tau \vdash e \,:\, \tau'}{\mathcal{E}; \Delta; \Gamma \vdash \lambda x{:}\tau.\, e \,:\, \tau \to \tau'}$$

$$\frac{\begin{array}{c}\mathcal{E}; \Delta; \Gamma, x{:}\tau \vdash v \,:\, \tau \\ \tau = \forall^{+} j_1 \ldots j_n.\, \forall \alpha_1{:}\kappa_1 \ldots \alpha_m{:}\kappa_m{:}\tau_1 \to \tau_2. \\ n \geq 0, m \geq 0\end{array}}{\mathcal{E}; \Delta; \Gamma \vdash \mathsf{fix}\, x{:}\tau.\, v \,:\, \tau}$$

$$\frac{\mathcal{E}; \Delta; \Gamma \vdash e \,:\, \forall^{+} \tau \qquad \mathcal{E} \vdash \kappa}{\mathcal{E}; \Delta; \Gamma \vdash e\, [\kappa]^{+} \,:\, \tau\, [\kappa]}$$

$$\frac{\mathcal{E}; \Delta; \Gamma \vdash e \,:\, \forall\, [\kappa]\, \tau \qquad \mathcal{E}; \Delta \vdash \tau' \,:\, \kappa}{\mathcal{E}; \Delta; \Gamma \vdash e\, [\tau'] \,:\, \tau\, \tau'}$$

$$\frac{\mathcal{E}; \Delta; \Gamma \vdash e \,:\, \tau' \to \tau \qquad \mathcal{E}; \Delta; \Gamma \vdash e' \,:\, \tau'}{\mathcal{E}; \Delta; \Gamma \vdash e\, e' \,:\, \tau}$$

$$\frac{\begin{array}{l}\mathcal{E}; \Delta \vdash \tau \;:\, \Omega \to \Omega \\ \mathcal{E}; \Delta \vdash \tau' \,:\, \Omega \\ \mathcal{E}; \Delta; \Gamma \vdash e_{\mathsf{int}} \,:\, \tau\, \mathsf{int} \\ \mathcal{E}; \Delta; \Gamma \vdash e_{\to} \,:\, \forall \alpha{:}\Omega.\, \forall \alpha'{:}\Omega.\, \tau\, (\alpha \to \alpha') \\ \mathcal{E}; \Delta; \Gamma \vdash e_{\forall} \;:\, \forall^{+} j.\, \forall \alpha{:}j \to \Omega.\, \tau\, (\forall\, [j]\, \alpha) \\ \mathcal{E}; \Delta; \Gamma \vdash e_{\forall^{+}} \,:\, \forall \alpha{:}(\forall j.\, \Omega).\, \tau\, (\forall^{+} \alpha)\end{array}}{\mathcal{E}; \Delta; \Gamma \vdash \mathsf{typecase}[\tau]\, \tau' \,\mathsf{of}\, (e_{\mathsf{int}};\, e_{\to};\, e_{\forall};\, e_{\forall^{+}}) \,:\, \tau\, \tau'}$$

Figure 3.5: Term formation rules of λ_i^{ω}

Type reduction $\quad \mathcal{E}; \Delta \vdash \tau_1 \rightsquigarrow \tau_2 : \kappa$

$$\frac{\mathcal{E}; \Delta, \alpha{:}\kappa' \vdash \tau : \kappa \quad \mathcal{E}; \Delta \vdash \tau' : \kappa'}{\mathcal{E}; \Delta \vdash (\lambda\alpha{:}\kappa'.\,\tau)\,\tau' \rightsquigarrow [\tau'/\alpha]\tau : \kappa}$$

$$\frac{\mathcal{E}, j; \Delta \vdash \tau : \forall j.\,\kappa \quad \mathcal{E} \vdash \kappa'}{\mathcal{E}; \Delta \vdash (\Lambda j.\,\tau)\,[\kappa'] \rightsquigarrow [\kappa'/j]\tau : [\kappa'/j]\kappa}$$

$$\frac{\mathcal{E}; \Delta \vdash \tau : \kappa \to \kappa' \quad \alpha \notin \mathit{ftv}(\tau)}{\mathcal{E}; \Delta \vdash \lambda\alpha{:}\kappa.\,\tau\,\alpha \rightsquigarrow \tau : \kappa \to \kappa'}$$

$$\frac{\mathcal{E}; \Delta \vdash \tau : \forall j'.\,\kappa \quad j \notin \mathit{fkv}(\tau)}{\mathcal{E}; \Delta \vdash \Lambda j.\,\tau\,[j] \rightsquigarrow \tau : \forall j'.\,\kappa}$$

$$\frac{\mathcal{E}; \Delta \vdash \mathsf{Typerec}[\kappa]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) : \kappa}{\mathcal{E}; \Delta \vdash \mathsf{Typerec}[\kappa]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \tau_{\mathsf{int}} : \kappa}$$

$$\frac{\begin{array}{c}\mathcal{E}; \Delta \vdash \mathsf{Typerec}[\kappa]\ \tau_1\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \tau_1' : \kappa \\ \mathcal{E}; \Delta \vdash \mathsf{Typerec}[\kappa]\ \tau_2\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \tau_2' : \kappa\end{array}}{\mathcal{E}; \Delta \vdash \mathsf{Typerec}[\kappa]\ ((\twoheadrightarrow)\,\tau_1\,\tau_2)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \tau_{\to}\,\tau_1\,\tau_2\,\tau_1'\,\tau_2' : \kappa}$$

$$\frac{\mathcal{E}; \Delta, \alpha{:}\kappa' \vdash \mathsf{Typerec}[\kappa]\ (\tau\,\alpha)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \tau' : \kappa}{\begin{array}{c}\mathcal{E}; \Delta \vdash \mathsf{Typerec}[\kappa]\ (\forall\,[\kappa']\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \\ \rightsquigarrow \tau_{\forall}\,[\kappa']\,\tau\,(\lambda\alpha{:}\kappa'.\,\tau') : \kappa\end{array}}$$

$$\frac{\mathcal{E}, j; \Delta \vdash \mathsf{Typerec}[\kappa]\ (\tau\,[j])\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \tau' : \kappa}{\mathcal{E}; \Delta \vdash \mathsf{Typerec}[\kappa]\ (\forall^+\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \tau_{\forall^+}\,\tau\,(\Lambda j.\,\tau') : \kappa}$$

Figure 3.6: λ_i^ω type reduction rules

judgements using the following environments:

$$\begin{array}{lrcl} \text{kind environment} & \mathcal{E} & ::= & \varepsilon \mid \mathcal{E}, j \\ \text{type environment} & \Delta & ::= & \varepsilon \mid \Delta, \alpha{:}\kappa \\ \text{term environment} & \Gamma & ::= & \varepsilon \mid \Gamma, x{:}\tau \end{array}$$

The Typerec operator analyzes polymorphic types with bound variables of arbitrary kind. The corresponding branch of the operator must bind the kind of the quantified type variable; for that purpose the language provides kind abstraction ($\Lambda j.\,\tau$) and kind application ($\tau\,[\kappa]$) at the type level. The formation rules for these constructs, excerpted from Figure 3.4, are

$$\frac{\mathcal{E}, j; \Delta \vdash \tau \,:\, \kappa}{\mathcal{E}; \Delta \vdash \Lambda j.\,\tau \,:\, \forall j.\,\kappa} \qquad \frac{\mathcal{E}; \Delta \vdash \tau \,:\, \forall j.\,\kappa \qquad \mathcal{E} \vdash \kappa'}{\mathcal{E}; \Delta \vdash \tau\,[\kappa'] \,:\, [\kappa'/j]\kappa}$$

Similarly, while analyzing a polymorphic type, the term-level construct typecase must bind the kind of the quantified type variable. Therefore, we introduce kind abstraction ($\Lambda^{+} j.\,e$) and kind application ($e\,[\kappa]^{+}$) at the term level. To type the term-level kind abstraction, we need a type construct $\forall^{+} j.\,\tau$ that binds the kind variable j in the type τ. The formation rules are shown below.

$$\frac{\mathcal{E}, j; \Delta; \Gamma \vdash v \,:\, \tau}{\mathcal{E}; \Delta; \Gamma \vdash \Lambda^{+} j.\,v \,:\, \forall^{+} j.\,\tau} \qquad \frac{\mathcal{E}; \Delta; \Gamma \vdash e \,:\, \forall^{+} j.\,\tau \quad \mathcal{E} \vdash \kappa}{\mathcal{E}; \Delta; \Gamma \vdash e\,[\kappa]^{+} \,:\, [\kappa/j]\tau}$$

However, since our goal is fully reflexive type analysis, we need to analyze kind-polymorphic types as well. As with polymorphic types, we can represent the type $\forall^{+} j.\,\tau$ as the application of a type constructor $\mathbb{V}^{+}$ of kind $(\forall j.\,\Omega) \to \Omega$ to a kind abstraction $\Lambda j.\,\tau$. Thus the kinds of the constructors for types of kind Ω are

$$\begin{array}{rcl} \mathsf{int} & : & \Omega \\ \twoheadrightarrow & : & \Omega \to \Omega \to \Omega \\ \mathbb{V} & : & \forall j.\,(j \to \Omega) \to \Omega \\ \mathbb{V}^{+} & : & (\forall j.\,\Omega) \to \Omega \end{array}$$

None of these constructors' arguments have the kind Ω in a negative position; hence the kind Ω can now be defined inductively in terms of these constructors. The Typerec construct is then the iterator over this inductive definition. The formation rule for Typerec follows naturally from the type reduction rules (Figure 3.6). Depending on the head constructor of the type being analyzed,

Typerec chooses one of the branches. At the int type, it returns the τ_{int} branch. At the function type $\tau \to \tau'$, it applies the $\tau_{\to}$ branch to the components τ and τ' and to the result of the iteration over τ and τ'.

When analyzing a polymorphic type, the reduction rule is

$$\begin{aligned}&\mathsf{Typerec}[\kappa]\ (\forall\alpha{:}\kappa'.\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow\\ &\tau_{\forall}\,[\kappa']\,(\lambda\alpha{:}\kappa'.\,\tau)\,(\lambda\alpha{:}\kappa'.\,\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}))\end{aligned}$$

Thus the $\forall$-branch of Typerec receives as arguments the kind of the bound variable, the abstraction representing the quantified type, and a type function encapsulating the result of the iteration on the body of the quantified type. Since $\tau_{\forall}$ must be parametric in the kind κ' (there are no facilities for kind analysis in the language), it can only apply its second and third arguments to locally introduced type variables of kind κ'. We believe this restriction, which is crucial for preserving strong normalization of the type language, is quite reasonable in practice. For instance $\tau_{\forall}$ can yield a quantified type based on the result of the iteration.

The reduction rule for analyzing a kind-polymorphic type is

$$\begin{aligned}&\mathsf{Typerec}[\kappa]\ (\forall^+ j.\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow\\ &\tau_{\forall^+}\,(\Lambda j.\,\tau)\,(\Lambda j.\,\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}))\end{aligned}$$

The arguments of the $\tau_{\forall^+}$ are the kind abstraction underlying the kind-polymorphic type and a kind abstraction encapsulating the result of the iteration on the body of the quantified type.

At the term level type analysis is carried out by the typecase construct; however, it is not iterative since the term language has a recursion primitive, fix. The $e_{\forall}$ branch of typecase binds the kind and the type abstraction carried by the type constructor $\forall$, while the $e_{\forall^+}$ branch binds the kind abstraction carried by $\forall^+$.

$$\begin{aligned}&\mathsf{typecase}[\tau]\ (\forall\,[\kappa]\,\tau')\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+}) \rightsquigarrow e_{\forall}\,[\kappa]^+\,[\tau']\\ &\mathsf{typecase}[\tau]\ (\forall^+\tau')\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+}) \rightsquigarrow e_{\forall^+}\,[\tau']\end{aligned}$$

The operational semantics of the term language of λ_i^ω is presented in Figure 3.7.

The language λ_i^ω has the following important properties. The proofs are given in Appendix A.

Theorem 3.3.1 *Reduction of well-formed types is strongly normalizing.*

$$(\lambda x\!:\!\tau.\, e)\, v \rightsquigarrow [v/x]e \qquad (\mathsf{fix}\, x\!:\!\tau.\, v)\, v' \rightsquigarrow ([\mathsf{fix}\, x\!:\!\tau.\, v/x]v)\, v'$$

$$(\Lambda\alpha\!:\!\kappa.\, v)[\tau] \rightsquigarrow [\tau/\alpha]v \qquad (\mathsf{fix}\, x\!:\!\tau.\, v)[\tau] \rightsquigarrow ([\mathsf{fix}\, x\!:\!\tau.\, v/x]v)[\tau]$$

$$(\Lambda^{+} j.\, v)[\kappa]^{+} \rightsquigarrow [\kappa/j]v \qquad (\mathsf{fix}\, x\!:\!\tau.\, v)[\kappa]^{+} \rightsquigarrow ([\mathsf{fix}\, x\!:\!\tau.\, v/x]v)[\kappa]^{+}$$

$$\frac{e \rightsquigarrow e'}{e\, e_1 \rightsquigarrow e'\, e_1} \qquad \frac{e \rightsquigarrow e'}{v\, e \rightsquigarrow v\, e'} \qquad \frac{e \rightsquigarrow e'}{e[\tau] \rightsquigarrow e'[\tau]} \qquad \frac{e \rightsquigarrow e'}{e[\kappa]^{+} \rightsquigarrow e'[\kappa]^{+}}$$

$$\mathsf{typecase}[\tau]\ \mathsf{int}\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}}) \rightsquigarrow e_{\mathsf{int}}$$

$$\mathsf{typecase}[\tau]\ (\tau_1 \rightarrow \tau_2)\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}}) \rightsquigarrow e_{\rightarrow}\, [\tau_1]\, [\tau_2]$$

$$\mathsf{typecase}[\tau]\ (\forall\, [\kappa]\, \tau)\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}}) \rightsquigarrow e_{\forall}\, [\kappa]^{+}\, [\tau]$$

$$\mathsf{typecase}[\tau]\ (\forall^{+} \tau)\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}}) \rightsquigarrow e_{\forall^{+}}\, [\tau]$$

$$\frac{\varepsilon;\varepsilon \vdash \tau' \rightsquigarrow^{*} \nu'\!:\!\Omega \qquad \nu' \text{ is a normal form}}{\begin{array}{c}\mathsf{typecase}[\tau]\ \tau'\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}}) \rightsquigarrow \\ \mathsf{typecase}[\tau]\ \nu'\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}})\end{array}}$$

Figure 3.7: Operational semantics of λ_i^{ω}

Theorem 3.3.2 *Reduction of well-formed types is confluent.*

Theorem 3.3.3 *If* $\varepsilon;\, \varepsilon;\, \varepsilon \vdash e\!:\!\tau$ *then either* e *is a value or there exists an* e' *such that* $e \rightsquigarrow e'$.

3.3.1 Example: Polymorphic equality

For ease of presentation, we will use ML-style pattern matching syntax to define a type involving Typerec. Instead of

$$\begin{aligned}
&\tau = \lambda\alpha\!:\!\Omega.\, \mathsf{Typerec}[\kappa]\ \alpha\ \mathsf{of}\ (\tau_{\mathsf{int}};\, \tau_{\rightarrow};\, \tau_{\forall};\, \tau_{\forall^{+}}) \\
&\text{where} \quad \tau_{\rightarrow} = \lambda\alpha_1\!:\!\Omega.\, \lambda\alpha_2\!:\!\Omega.\, \lambda\alpha'_1\!:\!\kappa.\, \lambda\alpha'_2\!:\!\kappa.\, \tau'_{\rightarrow} \\
&\qquad\qquad \tau_{\forall} = \Lambda j.\, \lambda\alpha\!:\!j \rightarrow \Omega.\, \lambda\alpha'\!:\!j \rightarrow \kappa.\, \tau'_{\forall} \\
&\qquad\qquad \tau_{\forall^{+}} = \lambda\alpha\!:\!(\forall j.\, \Omega).\, \lambda\alpha'\!:\!(\forall j.\, \kappa).\, \tau'_{\forall^{+}}
\end{aligned}$$

We will write

$$\begin{aligned}
\tau\,(\mathsf{int}) &= \tau_{\mathsf{int}} \\
\tau\,(\alpha_1 \to \alpha_2) &= [\tau\,(\alpha_1), \tau\,(\alpha_2)/\alpha'_1, \alpha'_2]\tau'_{\to} \\
\tau\,(\forall\,[j]\,\alpha_1) &= [\lambda\alpha\!:\!j.\,\tau\,(\alpha_1\,\alpha)/\alpha']\tau'_{\forall} \\
\tau\,(\forall^{+}\alpha_1) &= [\Lambda j.\,\tau\,(\alpha_1\,[j])/\alpha']\tau'_{\forall^{+}}
\end{aligned}$$

To illustrate the type-level analysis we will use the Typerec operator to define the class of types admitting equality comparisons [HM95]. To make the example non-trivial we extend the language with a product type constructor $\ltimes$ of the same kind as $\twoheadrightarrow$, and with existential types with type constructor $\exists$ of kind identical to that of $\forall$. We will write $\exists\alpha:\kappa.\,\tau$ for $\exists\,[\kappa]\,(\lambda\alpha:\kappa.\,\tau)$. Correspondingly we extend Typerec with a product branch $\tau_{\times}$ and an existential branch $\tau_{\exists}$ which behave in exactly the same way as the $\tau_{\to}$ branch and the $\tau_{\forall}$ branch respectively. We will use Bool instead of int.

A polymorphic function eq comparing two objects for equality is not defined on values of function or polymorphic types. We will enforce this restriction statically by defining a type operator Eq of kind $\Omega \to \Omega$, which maps function and polymorphic types to the type $\mathsf{Void} \equiv \forall\alpha:\Omega.\,\alpha$ (a type with no values), and require the arguments of eq to be of type $\mathsf{Eq}\,\tau$ for some type τ. Thus, given any type τ, the function Eq serves to verify that a non-equality type does not occur inside τ.

$$\begin{aligned}
\mathsf{Eq}\,(\mathsf{Bool}) &= \mathsf{Bool} \\
\mathsf{Eq}\,(\alpha_1 \to \alpha_2) &= \mathsf{Void} \\
\mathsf{Eq}\,(\alpha_1 \times \alpha_2) &= \mathsf{Eq}\,(\alpha_1) \times \mathsf{Eq}\,(\alpha_2) \\
\mathsf{Eq}\,(\forall\,[j]\,\alpha) &= \mathsf{Void} \\
\mathsf{Eq}\,(\forall^{+}\alpha) &= \mathsf{Void} \\
\mathsf{Eq}\,(\exists\,[j]\,\alpha) &= \exists\,[j]\,(\lambda\alpha_1:j.\,\mathsf{Eq}\,(\alpha\,\alpha_1))
\end{aligned}$$

The property is enforced even on hidden types in an existentially typed package by the reduction rule for Typerec which suspends its action on normal forms with a type variable at the head. For instance a term e can only be given a type $\mathsf{Eq}\,(\exists\alpha:\Omega.\,\alpha\times\alpha) = \exists\alpha:\Omega.\,\mathsf{Eq}\,\alpha \times \mathsf{Eq}\,\alpha$ if it can be shown that e is a pair of terms of type $\mathsf{Eq}\,\tau$ for some τ, *i.e.*, terms of equality type.

The term-level analysis of quantified types is illustrated by the polymorphic equality function. The term constructs for introduction and elimination of existential types have the usual formation

```
letrec
  heq : ∀α:Ω. ∀α′:Ω. Eq α → Eq α′ → Bool
  = Λα:Ω. Λα′:Ω.
      typecase[λγ:Ω. Eq γ → Eq α′ → Bool] α of
        Bool    ⇒ λx:Bool.
                    typecase[λγ:Ω. Eq γ → Bool] α′ of
                      Bool    ⇒ λy:Bool. primEqBool x y
                      ...     ⇒ ... false
        β1 × β2 ⇒ λx:Eq β1 × Eq β2.
                    typecase[λγ:Ω. Eq γ → Bool] α′ of
                      β1′ × β2′ ⇒ λy:Eq β1′ × Eq β2′.
                                    heq [β1] [β1′] (π1(x)) (π1(y)) and
                                    heq [β2] [β2′] (π2(x)) (π2(y))
                      ...       ⇒ ... false
        ∃[j] β  ⇒ λx:(∃β1:j. Eq (β β1)).
                    typecase[λγ:Ω. Eq γ → Bool] α′ of
                      ∃[j′] β′ ⇒ λy:(∃β1′:j′. Eq (β′ β1′)).
                                  open x as ⟨β1:j, xc:Eq (β β1)⟩ in
                                  open y as ⟨β1′:j′, yc:Eq (β′ β1′)⟩ in
                                    heq [β β1] [β′ β1′] xc yc
                      ...       ⇒ ... false
        ...
in let eq = Λα:Ω. λx:Eq α. λy:Eq α. heq [α] [α] x y
in ...
```

Figure 3.8: Polymorphic equality in λ_i^ω

rules:

$$\frac{\mathcal{E};\Delta;\Gamma \vdash e \,:\, (\lambda\alpha{:}\kappa.\,\tau)\,\tau'}{\mathcal{E};\Delta;\Gamma \vdash \langle \alpha{:}\kappa = \tau',\, e{:}\tau\rangle \,:\, \exists\alpha{:}\kappa.\,\tau} \qquad \frac{\mathcal{E};\Delta;\Gamma \vdash e \,:\, \exists\,[\kappa]\,\tau \qquad \mathcal{E};\Delta \vdash \tau' \,:\, \Omega \quad \mathcal{E};\Delta,\alpha{:}\kappa;\Gamma,x{:}\tau\,\alpha \vdash e' \,:\, \tau'}{\mathcal{E};\Delta;\Gamma \vdash \mathsf{open}\ e\ \mathsf{as}\ \langle \alpha{:}\kappa,\, x{:}\tau\,\alpha\rangle\ \mathsf{in}\ e' \,:\, \tau'}$$

The polymorphic equality function eq is defined in Figure 3.8 (the letrec construct can be derived from the fix). The domain type of the function is restricted to types of the form $\mathsf{Eq}\,\tau$ to ensure that only values of types admitting equality are compared.

Consider the two packages $v = \langle \alpha : \Omega = \mathsf{Bool}, \mathsf{false} : \alpha\rangle$ and $v' = \langle \alpha : \Omega = \mathsf{Bool} \times \mathsf{Bool}, \langle \mathsf{true}, \mathsf{true}\rangle : \alpha\rangle$. Both are of type $\exists\alpha:\Omega.\,\alpha$, which makes the invocation $\mathsf{eq}\,[\exists\alpha:\Omega.\,\alpha]\,v\,v'$ legal. But when the packages are open, the types of the packaged values may (as in this example) turn out to be different. Therefore we need the auxiliary function heq to compare values of possibly

different types by comparing their types first. The function corresponds to a matrix on the types of the two arguments, where the diagonal elements compare recursively the constituent values, while off-diagonal elements return false and are abbreviated in the figure.

The only interesting case is that of values of an existential type. Opening the packages provides access to the witness types β_1 and β_1' of the arguments x and y. As shown in the typing rules, the actual types of the packaged values, x and y, are obtained by applying the corresponding type functions β and β' to the respective witness types. This yields a perhaps unexpected semantics of equality. Consider this invocation of the eq function which evaluates to true:

$$\begin{aligned}&\mathsf{eq}\,[\exists\alpha{:}\Omega.\,\alpha]\\&\quad\langle\alpha{:}\Omega=\exists\beta{:}\Omega.\,\beta,\ \langle\beta{:}\Omega=\mathsf{Bool},\ \mathsf{true}:\mathsf{Eq}\,\beta\rangle:\mathsf{Eq}\,\alpha\rangle\\&\quad\langle\alpha{:}\Omega=\exists\beta{:}\Omega\to\Omega.\,\beta\,\mathsf{Bool},\ \langle\beta{:}\Omega\to\Omega=\lambda\gamma{:}\Omega.\,\gamma,\ \mathsf{true}:\mathsf{Eq}\,(\beta\,\mathsf{Bool})\rangle:\mathsf{Eq}\,\alpha\rangle\end{aligned}$$

At runtime, after the two packages are opened, the call to heq is

$$\begin{aligned}&\mathsf{heq}\,[\exists\beta{:}\Omega.\,\beta]\,[\exists\beta{:}\Omega\to\Omega.\,\beta\,\mathsf{Bool}]\\&\quad\langle\beta{:}\Omega=\mathsf{Bool},\ \mathsf{true}:\mathsf{Eq}\,\beta\rangle\\&\quad\langle\beta{:}\Omega\to\Omega=\lambda\gamma{:}\Omega.\,\gamma,\ \mathsf{true}:\mathsf{Eq}\,(\beta\,\mathsf{Bool})\rangle\end{aligned}$$

This term evaluates to true even though the type arguments are different. The reason is that what is being compared are the actual types of the values before hiding their witness types. Tracing the reduction of this term to the recursive call $\mathsf{heq}\,[\beta\,\beta_1]\,[\beta'\,\beta_1']\,\mathsf{xc}\,\mathsf{yc}$ we find out it is instantiated to

$$\mathsf{heq}\,[(\lambda\beta{:}\Omega.\,\beta)\,\mathsf{Bool}]\,[(\lambda\beta{:}\Omega\to\Omega.\,\beta\,\mathsf{Bool})\,(\lambda\gamma{:}\Omega.\,\gamma)]\,\mathsf{true}\,\mathsf{true}$$

which reduces to $\mathsf{heq}\,[\mathsf{Bool}]\,[\mathsf{Bool}]\,\mathsf{true}\,\mathsf{true}$ and thus to true.

This result is justified since the above two packages of type $\exists\alpha:\Omega.\,\alpha$ will indeed behave identically in all contexts. An informal argument in support of this claim is that the most any context could do with such a package is open it and inspect the type of its value using $\mathsf{typecase}$, but this will only provide access to a *type function* τ representing the inner existential type. Since the kind κ of the domain of τ is unknown statically, the only non-trivial operation on τ is its application to the witness type of the package, which is the only available type of kind κ. As we saw above, this operation will produce the same result (namely Bool) in both cases. Thus, since the

two arguments to eq are indistinguishable by λ_i^ω contexts, the above result is perfectly sensible.

3.3.2 Discussion

Before moving on, it would be worthwhile to analyze the λ_i^ω language. Specifically, what is the price in terms of complexity of the type theory that can be attributed to the requirements that were imposed?

In Section 3.2.2 we saw that an iterative type operator is essential to typechecking many type-directed operations. Even when restricted to compiling ML we still have to consider analysis of polymorphic types of the form $\forall\alpha{:}\Omega.\,\tau$, and their *ad hoc* inclusion in kind Ω makes the latter non-inductive. Therefore, even for this simple case, we need kind polymorphism in an essential way to handle the negative occurrence of Ω in the domain of $\forall$. In turn, kind polymorphism allows us to analyze at the type level types quantified over any kind; hence the extra expressiveness comes for free. Moreover, adding kind polymorphism does not entail any heavy type-theoretic machinery—the kind and type language of λ_i^ω is a minor extension (with primitive recursion) of the well-studied calculus F_2; we use the basic techniques developed for F_2 [GLT89] to prove properties of our type language.

The kind polymorphism of λ_i^ω is parametric, *i.e.*, kind analysis is not possible. This property prevents in particular the construction of non-terminating types based on variants of Girard's J operator using a kind-comparing operator [HM99].

For analysis of quantified types at the term level we have the new construct $\Lambda^+ j.\,e$. This does not result in any additional complexity at the type level—although we introduce a new type constructor $\forall^+$, the kind of this construct is defined completely by the original kind calculus, and the kind and type calculus is still essentially F_2. The term calculus becomes an extension of Girard's λU calculus [Gir72], hence it is not normalizing; however it already includes the general recursion construct fix, necessary in a realistic programming language.

Restricting the type analysis at the term level to a finite set of kinds would help avoid the term-level kind abstraction. However, even in this case, we would still need kind abstraction to implement a type erasure semantics. On the other hand, having kind abstraction at the term level of λ_i^ω adds no complications to the transition to type erasure semantics.

$$
\begin{array}{lll}
(\mathit{kinds}) & \kappa & ::= \Omega \mid \mathsf{T} \mid \kappa \to \kappa' \mid j \mid \forall j.\,\kappa \\
\\
(\mathit{types}) & \tau & ::= \mathsf{int} \mid \twoheadrightarrow \mid \mathbb{\forall} \mid \mathbb{\forall}^{+} \mid R \\
 & & \quad \mid T_{\mathsf{int}} \mid T_{\to} \mid T_{\forall} \mid T_{\forall^{+}} \mid T_{R} \\
 & & \quad \mid \alpha \mid \Lambda j.\,\tau \mid \tau\,[\kappa] \mid \lambda\alpha{:}\kappa.\,\tau \mid \tau\,\tau' \\
 & & \quad \mid \mathsf{Tagrec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^{+}};\ \tau_{R}) \\
(\mathit{values}) & v & ::= i \mid \Lambda^{+} j.\,v \mid \Lambda\alpha{:}\kappa.\,v \mid \lambda x{:}\tau.\,e \mid \mathsf{fix}\,x{:}\tau.\,v \\
 & & \quad \mid \mathsf{R}_{\mathsf{int}} \mid \mathsf{R}_{\to} \mid \mathsf{R}_{\to}\,[\tau] \mid \mathsf{R}_{\to}\,[\tau]\,v \\
 & & \quad \mid \mathsf{R}_{\to}\,[\tau]\,v\,[\tau'] \mid \mathsf{R}_{\to}\,[\tau]\,v\,[\tau']\,v' \\
 & & \quad \mid \mathsf{R}_{\forall} \mid \mathsf{R}_{\forall}\,[\kappa]^{+} \mid \mathsf{R}_{\forall}\,[\kappa]^{+}\,[\tau] \mid \mathsf{R}_{\forall}\,[\kappa]^{+}\,[\tau]\,[\tau'] \\
 & & \quad \mid \mathsf{R}_{\forall}\,[\kappa]^{+}\,[\tau]\,[\tau']\,v \\
 & & \quad \mid \mathsf{R}_{\forall^{+}} \mid \mathsf{R}_{\forall^{+}}\,[\tau] \mid \mathsf{R}_{\forall^{+}}\,[\tau]\,v \\
 & & \quad \mid \mathsf{R}_{R} \mid \mathsf{R}_{R}\,[\tau] \mid \mathsf{R}_{R}\,[\tau]\,v \\
\\
(\mathit{terms}) & e & ::= v \mid x \mid e\,[\kappa]^{+} \mid e\,[\tau] \mid e\,e' \\
 & & \quad \mid \mathsf{repcase}[\tau]\ e\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^{+}};\ e_{R})
\end{array}
$$

Figure 3.9: Syntax of the λ_R^ω language

3.4 Type erasure semantics

The main motivation for developing type systems for runtime type analysis is to use them in a type-based certifying compiler. Crary et al. [CWM98] proposed a framework that helps in propagating types through a type-preserving compiler. From an implementor's point of view, this framework (hereafter referred to as the CWM framework) seems to simplify some phases in a type preserving compiler; most notably, typed closure conversion [MMH96]. The main idea is to construct and pass terms representing types, instead of the types themselves, at runtime. This allows the use of pre-existing term operations to deal with runtime type information. Semantically, singleton types are used to connect a type to its representation.

However, the framework proposed in [CWM98] supports the analysis of inductively defined types only; it does not support the analysis of quantified types. In this section, we show how λ_i^ω can be translated into a type-erasure language. We call this language λ_R^ω. Figure 3.9 shows the syntax of the λ_R^ω language. To make the presentation simpler, we will describe many of the features in the context of the translation from λ_i^ω.

3.4.1 The analyzable components in λ_R^ω

In λ_R^ω, the type calculus is split into types and tags: While types classify terms, tags are used for analysis. We extend the kind calculus to distinguish between the two. The kind Ω includes the set of types, while the kind T includes the set of tags. For every constructor that generates a type of kind Ω, we have a corresponding constructor that generates a tag of kind T; for example, int corresponds to T_{int} and $\twoheadrightarrow$ corresponds to $T_{\rightarrow}$. The type analysis construct at the type level is Tagrec and operates only on tags.

At the term level, we add representations for tags. The term level operator (now called $\mathsf{repcase}$) analyzes these representations. All the primitive tags have corresponding term level representations; for example, T_{int} is represented by $\mathsf{R}_{\mathsf{int}}$. Given any tag, the corresponding term representation can be constructed inductively.

3.4.2 Typing term representations

The type calculus in λ_R^ω includes a unary type constructor R of kind $\mathsf{T} \rightarrow \Omega$ to type the term level representations. Given a tag τ (of kind T), the term representation of τ has the type $R\,\tau$. For example, $\mathsf{R}_{\mathsf{int}}$ has the type $R\,T_{\mathsf{int}}$. Semantically, $R\,\tau$ is interpreted as a singleton type that is inhabited only by the term representation of τ [CWM98].

If the tag τ is of a function kind $\kappa \rightarrow \kappa'$, then the term representation of τ is a polymorphic function from representations to representations:

$$R_{\kappa\rightarrow\kappa'}\,(\tau) \equiv \forall\beta{:}\kappa.\, R_\kappa\,(\beta) \rightarrow R_{\kappa'}\,(\tau\,\beta)$$

However a problem arises if τ is of a variable kind j. The only way of knowing the type of its representation R_j is to construct it when j is instantiated. Hence programs translated into λ_R^ω must be such that for every kind variable j in the program, a corresponding type variable α_j, representing the type of the term representation for a tag of kind j, is also available.

This is why we need to go beyond the CWM framework. Their source language does not include kind polymorphism; therefore, they can compute the type of all the representations statically. This is also the reason that we need to introduce a new set of primitive type constructors and split the type calculus into types and tags. Consider the $\forall$ and the $\forall^+$ type constructors in λ_i^ω. The $\forall$ constructor binds a kind κ. When it is translated into λ_R^ω, the translated constructor must also, in

$$|\Omega| = \mathsf{T} \qquad |\kappa \to \kappa'| = |\kappa| \to |\kappa'|$$
$$|j| = j \qquad |\forall j.\, \kappa| = \forall j.\, (j \to \Omega) \to |\kappa|$$

Figure 3.10: Translation of λ_i^ω kinds to λ_R^ω kinds

addition, bind a type of kind $\kappa \to \Omega$. Therefore, we need a new constructor $T_\forall$. Similarly, the $\forall^+$ constructor binds a type function of kind $\forall j.\, \Omega$. When it is translated into λ_R^ω, the translated constructor must bind a type function of kind $|\forall j.\, \Omega|$. (See Figure 3.10.) Therefore, we introduce a new constructor $T_{\forall^+}$. Furthermore, if we only have Ω as the primitive kind, it will no longer be inductive. (The inductive definition would break for $T_{\forall^+}$). Therefore, we introduce a new kind T (for tags), and allow analysis only over tags.

This leads us to the kind translation from λ_i^ω to λ_R^ω (Figure 3.10). Since the analyzable component of λ_R^ω is of kind T, the λ_i^ω kind Ω is mapped to T. The polymorphic kind $\forall j.\, \kappa$ is translated to $\forall j.\, (j \to \Omega) \to |\kappa|$. Note that every kind variable j must now have a corresponding type variable α_j. Given a tag of variable kind j, the type of its term representation is given by α_j. Since the type of a term is always of kind Ω, the variable α_j has the kind $j \to \Omega$.

Lemma 3.4.1 $|[\kappa'/j]\kappa| = [|\kappa'|/j]|\kappa|$

Proof By induction over the structure of κ. □

Figure 3.11 shows the function R_κ. Suppose τ is a λ_i^ω type of kind κ and $|\tau|$ is its translation into λ_R^ω. The function R_κ gives the type of the term representation of $|\tau|$. Since this function is used by the translation from λ_i^ω to λ_R^ω, it is defined by induction on λ_i^ω kinds.

Lemma 3.4.2 $[|\kappa'|, R_{\kappa'}/j', \alpha_{j'}](R_\kappa) = R_{[\kappa'/j']\kappa}$

Proof By induction over the structure of κ. □

The formation rules for tags are displayed in Figure 3.12. Since the translation maps λ_i^ω type constructors to these tags, a type constructor of kind κ is mapped to a corresponding tag of kind $|\kappa|$. Thus, while the $\forall$ type constructor has the kind $\forall j.\, (j \to \Omega) \to \Omega$, the $T_\forall$ tag has the kind $\forall j.\, (j \to \Omega) \to (j \to \mathsf{T}) \to \mathsf{T}$.

Figure 3.13 also shows the type of the term representation of the primitive type constructors. These types agree with the definition of the function R_κ; for example, the type of $\mathsf{R}_\to$ is $R_{\Omega\to\Omega\to\Omega}\,(T_\to)$. The term formation rules in Figure 3.13 use a tag interpretation function F that is

$$\frac{\mathcal{E} \vdash \Delta}{\mathcal{E};\Delta \vdash R_\Omega \equiv R \,:\, \mathsf{T} \to \Omega} \qquad \frac{\mathcal{E};\Delta \vdash \alpha_j \,:\, j \to \Omega}{\mathcal{E};\Delta \vdash R_j \equiv \alpha_j \,:\, j \to \Omega}$$

$$\frac{\mathcal{E};\Delta \vdash R_\kappa \equiv \tau \,:\, |\kappa| \to \Omega \qquad \mathcal{E};\Delta \vdash R_{\kappa'} \equiv \tau' \,:\, |\kappa'| \to \Omega}{\mathcal{E};\Delta \vdash R_{\kappa\to\kappa'} \equiv \lambda\alpha{:}|\kappa \to \kappa'|.\, \forall\beta{:}|\kappa|.\, \tau\,\beta \to \tau'\,(\alpha\,\beta) \,:\, |\kappa \to \kappa'| \to \Omega}$$

$$\frac{\mathcal{E}, j; \Delta, \alpha_j{:}j \to \Omega \vdash R_\kappa \equiv \tau \,:\, |\kappa| \to \Omega}{\mathcal{E};\Delta \vdash R_{\forall j.\,\kappa} \equiv \lambda\alpha{:}|\forall j.\,\kappa|.\, \forall^{+} j.\, \forall\alpha_j{:}j \to \Omega.\, \tau\,(\alpha\,[j]\,\alpha_j) \,:\, |\forall j.\,\kappa| \to \Omega}$$

Figure 3.11: Types of representations at higher kinds

explained in Section 3.4.4.

3.4.3 Tag analysis in λ^{ω}_{R}

We now consider the tag analysis constructs in more detail. The term level analysis is done by the repcase construct. Figure 3.13 and Figure 3.14 show its static and dynamic semantics respectively. The expression being analyzed must be of type $R\,\tau$; therefore, repcase always analyzes term representation of tags. Operationally, it examines the representation at the head, selects the corresponding branch, and passes the components of the representation to the selected branch. Thus the rule for analyzing the representation of a polymorphic type is

$$\mathsf{repcase}[\tau]\ \mathsf{R}_\forall\,[\kappa]^{+}\,[\tau_\kappa]\,[\tau']\,(e')\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_\forall;\ e_{\forall^+};\ e_R;\ e_\mu;\ e_{pl}) \rightsquigarrow\ e_\forall\,[\kappa]^{+}\,[\tau_\kappa]\,[\tau']\,(e')$$

Type level analysis is performed by the Tagrec construct. The language must be fully reflexive, so Tagrec includes an additional branch for the new type constructor T_{R}.

Figure 3.15 shows the reduction rules for the Tagrec, which are similar to the reduction rules for the source language Typerec: given a tag, it calls itself recursively on the components of the tag and then passes the result of the recursive calls, along with the original components, to the corresponding branch. Thus the reduction rule for the function tag is

$$\mathsf{Tagrec}[\kappa]\ (T_{\to}\,\tau\,\tau')\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_\forall;\ \tau_{\forall^+};\ \tau_{R}) \rightsquigarrow$$
$$\tau_{\to}\,\tau\,\tau'\,(\mathsf{Tagrec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_\forall;\ \tau_{\forall^+};\ \tau_{R}))\,(\mathsf{Tagrec}[\kappa]\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_\forall;\ \tau_{\forall^+};\ \tau_{R}))$$

Kind formation $\quad \mathcal{E} \vdash \kappa$

$$\mathcal{E} \vdash \mathsf{T}$$

Type formation $\quad \mathcal{E}; \Delta \vdash \tau \,:\, \kappa$

$$\frac{\mathcal{E} \vdash \Delta}{\begin{array}{l}
\mathcal{E}; \Delta \vdash R \;\; : \mathsf{T} \to \Omega \\
\mathcal{E}; \Delta \vdash T_{\mathsf{int}} : \mathsf{T} \\
\mathcal{E}; \Delta \vdash T_{\to} \;: \mathsf{T} \to \mathsf{T} \to \mathsf{T} \\
\mathcal{E}; \Delta \vdash T_{\forall} \;\; : \forall j.\, (j \to \Omega) \to (j \to \mathsf{T}) \to \mathsf{T} \\
\mathcal{E}; \Delta \vdash T_{\forall^+} : (\forall j.\, (j \to \Omega) \to \mathsf{T}) \to \mathsf{T} \\
\mathcal{E}; \Delta \vdash T_{R} \;\; : \mathsf{T} \to \mathsf{T}
\end{array}}$$

$$\frac{\begin{array}{l}
\mathcal{E}; \Delta \vdash \tau \;\; : \mathsf{T} \\
\mathcal{E}; \Delta \vdash \tau_{\mathsf{int}} : \kappa \\
\mathcal{E}; \Delta \vdash \tau_{\to} \;: \mathsf{T} \to \mathsf{T} \to \kappa \to \kappa \to \kappa \\
\mathcal{E}; \Delta \vdash \tau_{\forall} \;\; : \forall j.\, (j \to \Omega) \to (j \to \mathsf{T}) \to (j \to \kappa) \to \kappa \\
\mathcal{E}; \Delta \vdash \tau_{\forall^+} : (\forall j.\, (j \to \Omega) \to \mathsf{T}) \to (\forall j.\, (j \to \Omega) \to \kappa) \to \kappa \\
\mathcal{E}; \Delta \vdash \tau_{R} \;\; : \mathsf{T} \to \kappa \to \kappa
\end{array}}{\mathcal{E}; \Delta \vdash \mathsf{Tagrec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}) \,:\, \kappa}$$

Figure 3.12: Formation rules for the new type constructs in λ^{ω}_{R}

Term formation $\quad \mathcal{E};\Delta;\Gamma \vdash e : \tau$

$$\frac{\mathcal{E};\Delta \vdash \Gamma}{\begin{array}{l}
\mathcal{E};\Delta;\Gamma \vdash \mathsf{R}_{\mathsf{int}} : R\,T_{\mathsf{int}} \\
\mathcal{E};\Delta;\Gamma \vdash \mathsf{R}_{\rightarrow} : R_{\Omega\rightarrow\Omega\rightarrow\Omega}\,(T_{\rightarrow}) \\
\mathcal{E};\Delta;\Gamma \vdash \mathsf{R}_{\forall} : R_{\forall j.\,(j\rightarrow\Omega)\rightarrow\Omega}\,(T_{\forall}) \\
\mathcal{E};\Delta;\Gamma \vdash \mathsf{R}_{\forall^+} : R_{(\forall j.\,\Omega)\rightarrow\Omega}\,(T_{\forall^+}) \\
\mathcal{E};\Delta;\Gamma \vdash \mathsf{R}_{R} : R_{\Omega\rightarrow\Omega}\,(T_{R}) \\
\mathcal{E};\Delta \vdash \tau : \mathsf{T}\rightarrow\Omega \\
\mathcal{E};\Delta;\Gamma \vdash e : R\,\tau' \\
\mathcal{E};\Delta;\Gamma \vdash e_{\mathsf{int}} : \tau\,T_{\mathsf{int}} \\
\mathcal{E};\Delta;\Gamma \vdash e_{\rightarrow} : \forall\alpha_1{:}\mathsf{T}.\,R\,\alpha_1 \rightarrow \forall\alpha_2{:}\mathsf{T}.\,R\,\alpha_2 \rightarrow \tau\,(T_{\rightarrow}\,\alpha_1\,\alpha_2) \\
\mathcal{E};\Delta;\Gamma \vdash e_{\forall} : \forall^{+}j.\,\forall\alpha_j{:}j\rightarrow\Omega. \\
\qquad\qquad \forall\alpha{:}j\rightarrow\mathsf{T}.\,R_{j\rightarrow\Omega}\,(\alpha)\rightarrow\tau\,(T_{\forall}\,[j]\,\alpha_j\,\alpha) \\
\mathcal{E};\Delta;\Gamma \vdash e_{\forall^+} : \forall\alpha{:}\forall j.\,(j\rightarrow\Omega)\rightarrow\mathsf{T}.\,R_{\forall j.\,\Omega}\,(\alpha)\rightarrow\tau\,(T_{\forall^+}\,\alpha) \\
\mathcal{E};\Delta;\Gamma \vdash e_{R} : \forall\alpha{:}\mathsf{T}.\,R\,\alpha\rightarrow\tau\,(T_{R}\,\alpha)
\end{array}}{\mathcal{E};\Delta;\Gamma \vdash \mathsf{repcase}[\tau]\ e\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^+};\ e_R) : \tau\,\tau'}$$

Figure 3.13: Formation rules for the new term constructs in λ^{ω}_{R}

Similarly, the reduction for the polymorphic tag is

$$\begin{array}{l}
\mathsf{Tagrec}[\kappa]\ (T_{\forall}\,[\kappa]\,\tau_{\kappa}\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}) \rightsquigarrow \\
\tau_{\forall}\,[\kappa]\,\tau_{\kappa}\,\tau\,(\lambda\alpha{:}\kappa.\,\mathsf{Tagrec}[\kappa]\ (\tau\,\alpha)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}))
\end{array}$$

3.4.4 The tag interpretation function

Programs in λ^{ω}_{R} pass tags at runtime since only tags can be analyzed. However, abstractions and the fixpoint operator must still carry type information for type checking. Therefore, these annotations must use a function mapping tags to types. Since these annotations are always of kind Ω, this function must map tags of kind T to types of kind Ω. This implies that we can use an iterator over

$$(\lambda x\!:\!\tau.\, e)\, v \rightsquigarrow [v/x]e$$

$$(\mathsf{fix}\, x\!:\!\tau.\, v)\, v' \rightsquigarrow ([\mathsf{fix}\, x\!:\!\tau.\, v/x]v)\, v'$$

$$(\Lambda\alpha\!:\!\kappa.\, v)\, [\tau] \rightsquigarrow [\tau/\alpha]v$$

$$(\mathsf{fix}\, x\!:\!\tau.\, v)\, [\tau] \rightsquigarrow ([\mathsf{fix}\, x\!:\!\tau.\, v/x]v)\, [\tau]$$

$$(\Lambda^{+} j.\, v)\, [\kappa]^{+} \rightsquigarrow [\kappa/j]v$$

$$(\mathsf{fix}\, x\!:\!\tau.\, v)\, [\kappa]^{+} \rightsquigarrow ([\mathsf{fix}\, x\!:\!\tau.\, v/x]v)\, [\kappa]^{+}$$

$$\frac{e \rightsquigarrow e_1}{e\, e' \rightsquigarrow e_1\, e'} \qquad \frac{e \rightsquigarrow e_1}{v\, e \rightsquigarrow v\, e_1}$$

$$\frac{e \rightsquigarrow e_1}{e\, [\tau] \rightsquigarrow e_1\, [\tau]} \qquad \frac{e \rightsquigarrow e_1}{e\, [\kappa]^{+} \rightsquigarrow e_1\, [\kappa]^{+}}$$

$$\mathsf{repcase}[\tau]\ \mathsf{R}_{\mathsf{int}}\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}};\, e_R) \rightsquigarrow e_{\mathsf{int}}$$

$$\mathsf{repcase}[\tau]\ \mathsf{R}_{\forall^{+}}\, [\tau']\, (e')\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}};\, e_R) \rightsquigarrow e_{\forall^{+}}\, [\tau']\, (e')$$

$$\mathsf{repcase}[\tau]\ \mathsf{R}_{R}\, [\tau']\, (e')\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}};\, e_R) \rightsquigarrow e_{R}\, [\tau']\, (e')$$

$$\mathsf{repcase}[\tau]\ \mathsf{R}_{\rightarrow}\, [\tau_1]\, (e_1)\, [\tau_2]\, (e_2)\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}};\, e_R;\, e_{\mu};\, e_{pl}) \rightsquigarrow e_{\rightarrow}\, [\tau_1]\, (e_1)\, [\tau_2]\, (e_2)$$

$$\mathsf{repcase}[\tau]\ \mathsf{R}_{\forall}\, [\kappa]^{+}\, [\tau_\kappa]\, [\tau']\, (e')\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}};\, e_R;\, e_{\mu};\, e_{pl}) \rightsquigarrow e_{\forall}\, [\kappa]^{+}\, [\tau_\kappa]\, [\tau']\, (e')$$

$$\frac{e \rightsquigarrow e'}{\mathsf{repcase}[\tau]\ e\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}};\, e_R) \rightsquigarrow \mathsf{repcase}[\tau]\ e'\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}};\, e_R)}$$

Figure 3.14: Term reduction rules of λ_R^ω

$$\frac{\mathcal{E};\Delta \vdash \mathsf{Tagrec}[\kappa]\ T_{\mathsf{int}}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R})\ :\ \kappa}{\mathcal{E};\Delta \vdash \mathsf{Tagrec}[\kappa]\ T_{\mathsf{int}}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}) \rightsquigarrow \tau_{\mathsf{int}}\ :\ \kappa}$$

$$\frac{\begin{array}{c}\mathcal{E};\Delta \vdash \mathsf{Tagrec}[\kappa]\ \tau_1\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}) \rightsquigarrow \tau_1'\ :\ \kappa \\ \mathcal{E};\Delta \vdash \mathsf{Tagrec}[\kappa]\ \tau_2\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}) \rightsquigarrow \tau_2'\ :\ \kappa\end{array}}{\mathcal{E};\Delta \vdash \mathsf{Tagrec}[\kappa]\ (T_{\rightarrow}\,\tau_1\,\tau_2)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}) \rightsquigarrow \tau_{\rightarrow}\,\tau_1\,\tau_2\,\tau_1'\,\tau_2'\ :\ \kappa}$$

$$\frac{\mathcal{E};\Delta,\alpha{:}\kappa' \vdash \mathsf{Tagrec}[\kappa]\ (\tau_2\,\alpha)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}) \rightsquigarrow \tau'\ :\ \kappa}{\mathcal{E};\Delta \vdash \mathsf{Tagrec}[\kappa]\ (T_{\forall}\,[\kappa']\,\tau_1\,\tau_2)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}) \rightsquigarrow \tau_{\forall}\,[\kappa']\,\tau_1\,\tau_2\,(\lambda\alpha{:}\kappa'.\,\tau')\ :\ \kappa}$$

$$\frac{\mathcal{E},j;\Delta,\alpha_j{:}j \rightarrow \Omega \vdash \mathsf{Tagrec}[\kappa]\ (\tau\,[j]\,\alpha_j)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}) \rightsquigarrow \tau'\ :\ \kappa}{\mathcal{E};\Delta \vdash \mathsf{Tagrec}[\kappa]\ (T_{\forall^+}\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}) \rightsquigarrow \tau_{\forall^+}\,\tau\,(\Lambda j.\,\lambda\alpha_j{:}j \rightarrow \Omega.\,\tau')\ :\ \kappa}$$

$$\frac{\mathcal{E};\Delta \vdash \mathsf{Tagrec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}) \rightsquigarrow \tau'\ :\ \kappa}{\mathcal{E};\Delta \vdash \mathsf{Tagrec}[\kappa]\ (T_{R}\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}) \rightsquigarrow \tau_{R}\,\tau\,\tau'\ :\ \kappa}$$

Figure 3.15: Reduction rules for λ^{ω}_{R} Typerec

tags to define the function as follows (using the pattern matching syntax as before):

$$\begin{array}{lcl}\mathsf{F}\,(T_{\mathsf{int}}) & = & \mathsf{int} \\ \mathsf{F}\,(T_{\rightarrow}\,\alpha_1\,\alpha_2) & = & \mathsf{F}\,(\alpha_1) \rightarrow \mathsf{F}\,(\alpha_2) \\ \mathsf{F}\,(T_{\forall}\,[j]\,\alpha_j\,\alpha) & = & \forall\beta{:}j.\,\alpha_j\,\beta \rightarrow \mathsf{F}\,(\alpha\,\beta) \\ \mathsf{F}\,(T_{\forall^+}\,\alpha) & = & \forall j.\,\forall\alpha_j{:}j \rightarrow \Omega.\,\mathsf{F}\,(\alpha\,[j]\,\alpha_j) \\ \mathsf{F}\,(T_{R}\,\alpha) & = & \mathsf{int}\end{array}$$

The function F takes a type tree in the T kind space and converts it into the corresponding tree in the Ω kind space. Therefore, it converts the tag T_{int} to the type int. For the other tags, it recursively converts the components into the corresponding types. The branch for the T_{R} tag is bogus but of the correct kind. The language λ^{ω}_{R} is only intended as a target for translation from λ^{ω}_{i}—the only interesting programs in λ^{ω}_{R} are the ones translated from λ^{ω}_{i}; therefore, the T_{R} branch of F will remain unused.

The type interpretation function has the following properties.

Lemma 3.4.3 $[\tau'/\alpha](\mathsf{F}\,(\tau)) = \mathsf{F}\,([\tau'/\alpha]\tau)$

Proof Follows from the fact that none of the branches of F has free type variables. □

Lemma 3.4.4 $[\kappa/j](\mathsf{F}(\tau)) = \mathsf{F}([\kappa/j]\tau)$

Proof Follows from the fact that none of the branches of F has free kind variables. □

The language λ_R^ω has the following properties. The proofs are the same as the proofs for the corresponding properties of λ_i^ω which are shown in Appendix A.

Proposition 3.4.5 (Type Reduction) *Reduction of well formed types is strongly normalizing and confluent.*

Proposition 3.4.6 (Type Safety) *If* $\vdash e : \tau$*, then either* e *is a value, or there exists a term* e' *such that* $e \rightsquigarrow e'$ *and* $\vdash e' : \tau$*.*

3.5 Translation from λ_i^ω to λ_R^ω

In this section, we show a translation from λ_i^ω to λ_R^ω. The languages differ mainly in two ways. First, the type calculus in λ_R^ω is split into tags and types, with types used solely for type checking and tags used for analysis. Therefore, type passing in λ_i^ω will get converted into tag passing in λ_R^ω. Second, the typecase operator in λ_i^ω must be converted into a repcase operating on term representation of tags.

Figure 3.16 shows the translation of λ_i^ω types into λ_R^ω tags. The primitive type constructors get translated into the corresponding tag constructors. The Typerec gets converted into a Tagrec. The translation inserts an arbitrarily chosen well-kinded result into the branch for the T_R tag since the source contains no such branch.

The term translation is shown in Figure 3.17. The translation must maintain two invariants. First, for every accessible kind variable j, it adds a corresponding type variable α_j; this variable gives the type of the term representation for a tag of kind j. At every kind application, the translation uses the function R_κ (Figure 3.11) to compute this type. Thus, the translations of kind abstractions and kind applications are

$$|\Lambda^+ j.\, v| = \Lambda^+ j.\, \Lambda\alpha_j : j \to \Omega.\, |v| \qquad |e\,[\kappa]^+| = |e|\,[|\kappa|]^+\,[R_\kappa]$$

Second, for every accessible type variable α, a term variable x_α is introduced, providing the corresponding term representation of α. At every type application, the translation uses the function

$$|\alpha| = \alpha$$

$$|\mathsf{int}| = T_{\mathsf{int}} \qquad |\Lambda j.\,\tau| = \Lambda j.\,\lambda\alpha_j : j \to \Omega.\,|\tau|$$

$$|\twoheadrightarrow| = T_{\to} \qquad |\tau\,[\kappa]| = |\tau|\,[|\kappa|]\,R_\kappa$$

$$|\forall| = T_{\forall} \qquad |\lambda\alpha:\kappa.\,\tau| = \lambda\alpha:|\kappa|.\,|\tau|$$

$$|\forall^+| = T_{\forall^+} \qquad |\tau\,\tau'| = |\tau|\,|\tau'|$$

$$|\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+})| =$$
$$\mathsf{Tagrec}[|\kappa|]\ |\tau|\ \mathsf{of}\ (|\tau_{\mathsf{int}}|;\ |\tau_{\to}|;\ |\tau_{\forall}|;\ |\tau_{\forall^+}|;\ \lambda\_:\mathsf{T}.\,\lambda\_:|\kappa|.\,|\tau_{\mathsf{int}}|)$$

Figure 3.16: Translation of λ_i^ω types to λ_R^ω tags

$$|i| = i$$

$$|x| = x$$

$$|\Lambda^+ j.\,v| = \Lambda^+ j.\,\Lambda\alpha_j : j \to \Omega.\,|v|$$

$$|e\,[\kappa]^+| = |e|\,[|\kappa|]^+\,[R_\kappa]$$

$$|\Lambda\alpha:\kappa.\,v| = \Lambda\alpha:|\kappa|.\,\lambda x_\alpha : R_\kappa\,\alpha.\,|v|$$

$$|e\,[\tau]| = |e|\,[|\tau|]\,\Re(\tau)$$

$$|\lambda x:\tau.\,e| = \lambda x:\mathsf{F}\,|\tau|.\,|e|$$

$$|e\,e'| = |e|\,|e'|$$

$$|\mathsf{fix}\,x:\tau.\,v| = \mathsf{fix}\,x:\mathsf{F}\,|\tau|.\,|v|$$

$$\begin{aligned}
&|\mathsf{typecase}[\tau]\ \tau'\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+})| \\
&\quad = \mathsf{repcase}[\lambda\alpha:\mathsf{T}.\,\mathsf{F}\,(|\tau|\,\alpha)]\ \Re(\tau')\ \mathsf{of} \\
&\qquad \mathsf{R}_{\mathsf{int}} \Rightarrow |e_{\mathsf{int}}| \\
&\qquad \mathsf{R}_{\to} \Rightarrow |e_{\to}| \\
&\qquad \mathsf{R}_{\forall} \Rightarrow |e_{\forall}| \\
&\qquad \mathsf{R}_{\forall^+} \Rightarrow |e_{\forall^+}| \\
&\qquad \mathsf{R}_R \Rightarrow \Lambda\beta:\mathsf{T}.\,\lambda x:R\,\beta.\,\mathsf{fix}\,x:\mathsf{F}\,(|\tau|\,(T_R\,\beta)).\,x
\end{aligned}$$

Figure 3.17: Translation of λ_i^ω terms to λ_R^ω terms

$$
\begin{aligned}
\Re(\mathsf{int}) &= \mathsf{R}_{\mathsf{int}} \\
\Re(\rightarrow) &= \Lambda\alpha{:}\mathsf{T}.\,\lambda x_\alpha : R\,\alpha.\,\Lambda\beta{:}\mathsf{T}.\,\lambda x_\beta : R\,\beta. \\
&\qquad \mathsf{R}_{\rightarrow}\,[\alpha]\,(x_\alpha)\,[\beta]\,(x_\beta) \\
\Re(\forall) &= \Lambda^{+} j.\,\Lambda\alpha_j : j \rightarrow \Omega.\,\Lambda\alpha : j \rightarrow \mathsf{T}.\,\lambda x_\alpha : R_{j\rightarrow\Omega}\,(\alpha). \\
&\qquad \mathsf{R}_{\forall}\,[j]^{+}\,[\alpha_j]\,[\alpha]\,(x_\alpha) \\
\Re(\forall^{+}) &= \Lambda\alpha : (\forall j.\,(j \rightarrow \Omega) \rightarrow \mathsf{T}).\,\lambda x_\alpha : R_{\forall j.\,\Omega}\,(\alpha). \\
&\qquad \mathsf{R}_{\forall^{+}}\,[\alpha]\,(x_\alpha) \\
\Re(\alpha) &= x_\alpha \\
\Re(\Lambda j.\,\tau) &= \Lambda^{+} j.\,\Lambda\alpha_j : j \rightarrow \Omega.\,\Re(\tau) \\
\Re(\tau\,[\kappa]) &= \Re(\tau)\,[|\kappa|]^{+}\,[R_\kappa] \\
\Re(\lambda\alpha : \kappa.\,\tau) &= \Lambda\alpha : |\kappa|.\,\lambda x_\alpha : R_\kappa\,\alpha.\,\Re(\tau) \\
\Re(\tau\,\tau') &= \Re(\tau)\,[|\tau'|]\,(\Re(\tau'))
\end{aligned}
$$

$$
\begin{aligned}
&\Re(\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^{+}})) \ = \\
&(\mathsf{fix}\,\mathsf{f} : \forall\alpha{:}\mathsf{T}.\,R\,\alpha \rightarrow R\ (\tau^{*}\,\alpha). \\
&\quad \Lambda\alpha{:}\mathsf{T}.\,\lambda x_\alpha : R\,\alpha. \\
&\quad \mathsf{repcase}[\lambda\alpha{:}\mathsf{T}.\,R\ (\tau^{*}\,\alpha)]\ x_\alpha\ \mathsf{of} \\
&\qquad \mathsf{R}_{\mathsf{int}} \Rightarrow \Re(\tau_{\mathsf{int}}) \\
&\qquad \mathsf{R}_{\rightarrow} \Rightarrow \Lambda\alpha{:}\mathsf{T}.\,\lambda x_\alpha : R\,\alpha.\,\Lambda\beta{:}\mathsf{T}.\,\lambda x_\beta : R\,\beta. \\
&\qquad\qquad \Re(\tau_{\rightarrow})\,[\alpha]\,(x_\alpha)\,[\beta]\,(x_\beta) \\
&\qquad\qquad\quad [\tau^{*}\,\alpha]\,(\mathsf{f}\,[\alpha]\,x_\alpha)\,[\tau^{*}\,\beta]\,(\mathsf{f}\,[\beta]\,x_\beta) \\
&\qquad \mathsf{R}_{\forall} \Rightarrow \Lambda^{+} j.\,\Lambda\alpha_j : j \rightarrow \Omega.\,\Lambda\alpha : j \rightarrow \mathsf{T}.\,\lambda x_\alpha : R_{j\rightarrow\Omega}\,(\alpha). \\
&\qquad\qquad \Re(\tau_{\forall})\,[j]^{+}\,[\alpha_j]\,[\alpha]\,(x_\alpha)\,[\lambda\beta : j.\,\tau^{*}\,(\alpha\,\beta)] \\
&\qquad\qquad\quad (\Lambda\beta : j.\,\lambda x_\beta : \alpha_j\,\beta.\,\mathsf{f}\,[\alpha\,\beta]\,(x_\alpha\,[\beta]\,x_\beta)) \\
&\qquad \mathsf{R}_{\forall^{+}} \Rightarrow \Lambda\alpha : (\forall j.\,(j \rightarrow \Omega) \rightarrow \mathsf{T}).\,\lambda x_\alpha : R_{\forall j.\,\Omega}\,(\alpha). \\
&\qquad\qquad \Re(\tau_{\forall^{+}})\,[\alpha]\,(x_\alpha)\,[\Lambda j.\,\lambda\alpha_j : j \rightarrow \Omega.\,\tau^{*}\,(\alpha\,[j]\,\alpha_j)] \\
&\qquad\qquad\quad (\Lambda^{+} j.\,\Lambda\alpha_j : j \rightarrow \Omega.\,\mathsf{f}\,[\alpha\,[j]\,\alpha_j]\,(x_\alpha\,[j]^{+}\,[\alpha_j])) \\
&\qquad \mathsf{R}_{R} \Rightarrow \Lambda\alpha{:}\mathsf{T}.\,\lambda x_\alpha : R\,\alpha.\,\Re(\tau_{\mathsf{int}}) \\
&)\,[|\tau|]\,\Re(\tau) \\
&\text{where} \\
&\quad \tau^{*} = |\lambda\alpha{:}\Omega.\,\mathsf{Typerec}[\kappa]\ \alpha\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^{+}})|
\end{aligned}
$$

Figure 3.18: Representation of λ_i^ω types as λ_R^ω terms

$\Re(\tau)$ (Figure 3.18) to construct this representation. Furthermore, type application gets replaced by an application to a tag, and to the term representation of the tag. Thus the translations for type abstractions and type applications are

$$|\Lambda\alpha{:}\kappa.\, v| = \Lambda\alpha{:}|\kappa|.\, \lambda x_\alpha : R_\kappa\, \alpha.\, |v| \qquad\qquad |e\,[\tau]| = |e|\,[|\tau|]\,\Re(\tau)$$

As pointed out before, the translations of abstraction and the fixpoint operator use the tag interpretation function F to map tags to types.

We show the term representation of types in Figure 3.18. The primitive type constructors get translated to the corresponding term representation. The representations of type and kind functions are similar to the term translation of type and kind abstractions. The only involved case is the term representation of a Typerec. Since Typerec is recursive, we use a combination of a repcase and a fix. We will illustrate only one case here; the other cases can be reasoned about similarly.

Consider the reduction of $\mathsf{Ty}\,(\tau' \to \tau'')$. ($\mathsf{Ty}\,\tau$ stands for $\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+})$) This type reduces to $\tau_{\to}\,\tau'\,\tau''\,(\mathsf{Ty}\,(\tau'))\,(\mathsf{Ty}\,(\tau''))$. Therefore, in the translation, the term representation of $\tau_{\to}$ must be applied to the term representations of τ', τ'', and the result of the recursive calls to the Typerec. The representations of τ' and τ'' are bound to the variables x_α and x_β; by assumption the representations for the results of the recursive calls are obtained from the recursive calls to the function f.

In the following propositions the original λ_i^ω kind environment Δ is extended with a kind environment $\Delta(\mathcal{E})$ which binds a type variable α_j of kind $j \to \Omega$ for each $j \in \mathcal{E}$. Similarly the term-level translations extend the type environment Γ with $\Gamma(\Delta)$, binding a variable x_α of type $R_\kappa\,\alpha$ for each type variable α bound in Δ with kind κ.

Proposition 3.5.1 *If* $\mathcal{E}; \Delta \vdash \tau\ :\ \kappa$ *holds in* λ_i^ω*, then* $|\mathcal{E}|; |\Delta|,\ \Delta(\mathcal{E}) \vdash |\tau|\ :\ |\kappa|$ *holds in* λ_R^ω.

Proof Follows directly by induction over the structure of τ. □

Proposition 3.5.2 *If* $\mathcal{E}; \Delta \vdash \tau\ :\ \kappa$ *and* $\mathcal{E}; \Delta \vdash \Gamma$ *hold in* λ_i^ω*, then* $|\mathcal{E}|; |\Delta|,\ \Delta(\mathcal{E}); |\Gamma|,\ \Gamma(\Delta) \vdash \Re(\tau)\ :\ R_\kappa\,|\tau|$ *holds in* λ_R^ω.

Proof By induction over the structure of τ. The only interesting case is that of a kind application which uses Lemma 3.4.2. □

$$
\begin{array}{lll}
(values) & v & ::= i \mid \lambda x.e \mid \mathsf{fix}\, x.v \\
& & \mid \mathsf{R}_{\mathsf{int}} \mid \mathsf{R}_{\rightarrow} \mid \mathsf{R}_{\rightarrow}\,\mathsf{1} \mid \mathsf{R}_{\rightarrow}\,\mathsf{1}\,v \\
& & \mid \mathsf{R}_{\rightarrow}\,\mathsf{1}\,v\,\mathsf{1} \mid \mathsf{R}_{\rightarrow}\,\mathsf{1}\,v\,\mathsf{1}\,v' \\
& & \mid \mathsf{R}_{\forall} \mid \mathsf{R}_{\forall}\,\mathsf{1} \mid \mathsf{R}_{\forall}\,\mathsf{1}\,\mathsf{1} \mid \mathsf{R}_{\forall}\,\mathsf{1}\,\mathsf{1}\,\mathsf{1} \\
& & \mid \mathsf{R}_{\forall}\,\mathsf{1}\,\mathsf{1}\,\mathsf{1}\,v \\
& & \mid \mathsf{R}_{\forall^+} \mid \mathsf{R}_{\forall^+}\,\mathsf{1} \mid \mathsf{R}_{\forall^+}\,\mathsf{1}\,v \\
& & \mid \mathsf{R}_{\mu} \mid \mathsf{R}_{\mu}\,\mathsf{1} \mid \mathsf{R}_{\mu}\,\mathsf{1}\,v \\
& & \mid \mathsf{R}_{pl} \mid \mathsf{R}_{pl}\,\mathsf{1} \mid \mathsf{R}_{pl}\,\mathsf{1}\,v \\
& & \mid \mathsf{R}_{R} \mid \mathsf{R}_{R}\,\mathsf{1} \mid \mathsf{R}_{R}\,\mathsf{1}\,v \\
\\
(terms) & e & ::= v \mid x \mid e\,e' \\
& & \mid \mathsf{repcase}\ e\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^+};\ e_R)
\end{array}
$$

Figure 3.19: Syntax of the untyped language $\lambda_R^{\omega\circ}$

Proposition 3.5.3 *If* $\mathcal{E};\Delta;\Gamma \vdash e : \tau$ *holds in* λ_i^ω*, then* $|\mathcal{E}|;|\Delta|,\ \Delta(\mathcal{E});|\Gamma|,\ \Gamma(\Delta) \vdash |e| : \mathsf{F}\,|\tau|$ *holds in* λ_R^ω.

Proof This is proved by induction over the structure of e, using Lemmas 3.4.3 and 3.4.4. □

3.6 The untyped language

This section shows that in λ_R^ω types are not necessary for computation. Figure 3.19 shows an untyped language $\lambda_R^{\omega\circ}$. We show a translation from λ_R^ω to $\lambda_R^{\omega\circ}$ in Figure 3.20. The expression $\mathsf{1}$ is the integer constant one.

The translation replaces type and kind applications (abstractions) by a dummy application (abstraction), instead of erasing them. In the typed language, a type or a kind can be applied to a fixpoint. This results in an unfolding of the fixpoint. Therefore, the translation inserts dummy applications to preserve this unfolding.

The untyped language has the following property which shows that term reduction in $\lambda_R^{\omega\circ}$ parallels term reduction in λ_R^ω.

Proposition 3.6.1 *If* $e \rightsquigarrow^* e_1$*, then* $e^\circ \rightsquigarrow^* {e_1}^\circ$.

Proof Follows by induction over the reduction relation. □

$$i^\circ = i$$

$$(\Lambda^+ j.\, v)^\circ = \lambda\_.v^\circ$$

$$(\Lambda\alpha\!:\!\kappa.\, v)^\circ = \lambda\_.v^\circ$$

$$(\lambda x\!:\!\tau.\, e)^\circ = \lambda x.e^\circ$$

$$(\mathsf{fix}\, x\!:\!\tau.\, v)^\circ = \mathsf{fix}\, x.v^\circ$$

$$(e\,[\kappa]^+)^\circ = e^\circ\,\mathsf{1}$$

$$(e\,[\tau])^\circ = e^\circ\,\mathsf{1}$$

$$(e\,e_1)^\circ = e^\circ\,{e_1}^\circ$$

$${\mathsf{R}_{\mathsf{int}}}^\circ = \mathsf{R}_{\mathsf{int}}$$

$${\mathsf{R}_{\to}}^\circ = \mathsf{R}_{\to}$$

$$(\mathsf{R}_{\to}\,[\tau])^\circ = \mathsf{R}_{\to}\,\mathsf{1}$$

$$(\mathsf{R}_{\to}\,[\tau]\,e)^\circ = \mathsf{R}_{\to}\,\mathsf{1}\,e^\circ$$

$$(\mathsf{R}_{\to}\,[\tau]\,e\,[\tau']\,)^\circ = \mathsf{R}_{\to}\,\mathsf{1}\,e^\circ\,\mathsf{1}$$

$$(\mathsf{R}_{\to}\,[\tau]\,e\,[\tau']\,e_1)^\circ = \mathsf{R}_{\to}\,\mathsf{1}\,e^\circ\,\mathsf{1}\,{e_1}^\circ$$

$${\mathsf{R}_{\forall}}^\circ = \mathsf{R}_{\forall}$$

$$(\mathsf{R}_{\forall}\,[\kappa]^+)^\circ = \mathsf{R}_{\forall}\,\mathsf{1}$$

$$(\mathsf{R}_{\forall}\,[\kappa]^+\,[\tau])^\circ = \mathsf{R}_{\forall}\,\mathsf{1}\,\mathsf{1}$$

$$(\mathsf{R}_{\forall}\,[\kappa]^+\,[\tau]\,[\tau']\,)^\circ = \mathsf{R}_{\forall}\,\mathsf{1}\,\mathsf{1}\,\mathsf{1}$$

$$(\mathsf{R}_{\forall}\,[\kappa]^+\,[\tau]\,[\tau']\,e)^\circ = \mathsf{R}_{\forall}\,\mathsf{1}\,\mathsf{1}\,\mathsf{1}\,e^\circ$$

$${\mathsf{R}_{\forall^+}}^\circ = \mathsf{R}_{\forall^+}$$

$$(\mathsf{R}_{\forall^+}\,[\tau])^\circ = \mathsf{R}_{\forall^+}\,\mathsf{1}$$

$$(\mathsf{R}_{\forall^+}\,[\tau]\,e)^\circ = \mathsf{R}_{\forall^+}\,\mathsf{1}\,e^\circ$$

$${\mathsf{R}_{\mu}}^\circ = \mathsf{R}_{\mu}$$

$$(\mathsf{R}_{\mu}\,[\tau])^\circ = \mathsf{R}_{\mu}\,\mathsf{1}$$

$$(\mathsf{R}_{\mu}\,[\tau]\,e)^\circ = \mathsf{R}_{\mu}\,\mathsf{1}\,e^\circ$$

$${\mathsf{R}_{pl}}^\circ = \mathsf{R}_{pl}$$

$$(\mathsf{R}_{pl}\,[\tau])^\circ = \mathsf{R}_{pl}\,\mathsf{1}$$

$$(\mathsf{R}_{pl}\,[\tau]\,e)^\circ = \mathsf{R}_{pl}\,\mathsf{1}\,e^\circ$$

$${\mathsf{R}_{R}}^\circ = \mathsf{R}_{R}$$

$$(\mathsf{R}_{R}\,[\tau])^\circ = \mathsf{R}_{R}\,\mathsf{1}$$

$$(\mathsf{R}_{R}\,[\tau]\,e)^\circ = \mathsf{R}_{R}\,\mathsf{1}\,e^\circ$$

$$(\mathsf{repcase}[\tau]\ e\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_R))^\circ\ =\ \mathsf{repcase}\ e^\circ\ \mathsf{of}\ ({e_{\mathsf{int}}}^\circ;\ {e_{\to}}^\circ;\ {e_{\forall}}^\circ;\ {e_{\forall^+}}^\circ;\ {e_R}^\circ)$$

Figure 3.20: Translation of λ_R^ω to ${\lambda_R^\omega}^\circ$

3.7 Related work

The work of Harper and Morrisett [HM95] introduced intensional type analysis and pointed out the necessity for type-level type analysis operators which inductively traverse the structure of types. The domain of their analysis is restricted to a predicative subset of the type language, which prevents its use in programs which must support all types of values, including polymorphic functions, closures, and objects. This chapter builds on their work by extending type analysis to include the full type language. Crary *et al.* [CW99] propose a very powerful type analysis framework. They define a rich kind calculus that includes sum kinds and inductive kinds. They also provide primitive recursion at the type level. Therefore, they can define new kinds within their calculus and directly encode type analysis operators within their language. They also include a novel refinement operation at the term level. However, their type analysis is "limited to parametrically polymorphic functions, and cannot account for functions that perform intensional type analysis" [CW99, Section 4.1]. The type analysis presented here can also handle polymorphic functions that analyze the quantified type variable. Moreover, their type analysis is not fully reflexive since they can not handle arbitrary quantified types; quantification must be restricted to type variables of kind Ω. Duggan [Dug98] proposes another framework for intensional type analysis; however, he allows the analysis of types only at the term level and not at the type level. Yang [Yan98] presents some approaches to enable type-safe programming of type-indexed values in ML which is similar to term-level analysis of types.

The idea of programming with iterators is explained in Pierce's notes [PDM89]. Pfenning and Mohring [PL89] show how inductively defined types can be represented by closed types. They also construct representations of all primitive recursive functions over inductively defined types.

The work on type-erasure semantics uses the framework proposed in Crary et al. [CWM98]. However, as we pointed out before, they consider a language that analyzes inductively defined types only. Extending the analysis to arbitrary types makes the translation much more complicated. The splitting of the type calculus into types and tags, and defining an interpretation function to map between the two, is somewhat related to the ideas proposed by Crary and Weirich for the language LX [CW99].

The erasure framework also resembles the dictionary passing style in Haskell [PJ93]. The term representation of a type may be viewed as the dictionary corresponding to the type. However, the authors consider dictionary passing in an untyped calculus; moreover, they do not consider

the intensional analysis of types. Dubois et al. [DRW95] also pass explicit type representations in their extensional polymorphism scheme. However, they do not provide a mechanism for connecting a type to its representation. Minamide's [Min97] type-lifting procedure is also related to this work. His procedure maintains interrelated constraints between type parameters; however, his language does not support intensional type analysis. Aspinall [Asp95] studied a typed λ-calculus with subtypes and singleton types.

Chapter 4

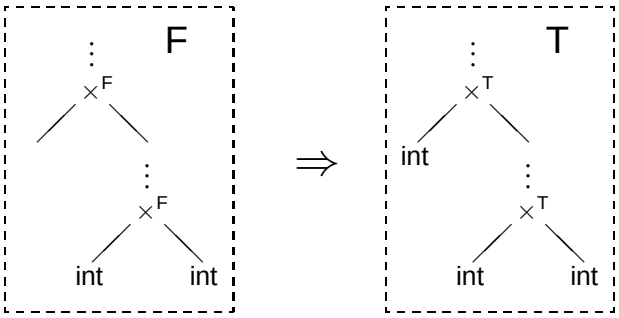
Applying Runtime Type Analysis: Garbage Collection

4.1 Introduction and motivation

In this chapter, we use fully reflexive type analysis for building a type-safe garbage collector (GC). We show that analysis of quantified types is crucial for accurately modelling the contract between the collector and the mutator. Type-safe GC is important because most type-safe systems rely critically on the type-safety of an underlying garbage collector. This also holds for Proof-Carrying Code (PCC) [Nec97] and Typed Assembly Languages (TAL) [MWCG98]. Indeed, constructing a verifiably type-safe garbage collector is widely considered as one of the major open problems in the area of certifying compilation [Mor00, Cra00].

A type-safe GC is not only desirable from the point of view of safety, but also for software-engineering purposes. A type-safe GC must make explicit the contract between the collector and the mutator, and type-checking makes sure that this contract is always respected. This can also aid in the process of choosing between GC variants without risking the integrity of the system. Writing GC inside a type-safe language also makes it possible to achieve principled interoperation between garbage collection and other memory-management mechanisms (e.g., those based on malloc-free and regions).

Recently, Wang and Appel [WA01] proposed to tackle the problem by building a tracing garbage collector on top of a region-based calculus, thus providing both type safety and completely automatic memory management. Their approach relies on monomorphization and defunctionaliza-



$copy : \forall \mathsf{F}.\forall \mathsf{T}.\forall \alpha.(\alpha \rightarrow [\mathsf{T}/\mathsf{F}]\alpha)$

Figure 4.1: Stop-and-Copy from region F to region T.

tion (a form of closure conversion due to Tolmach [TO98]). Our region calculus is similar to Wang and Appel, but we use runtime type analysis instead of monomorphization. The system presented in this chapter makes the following new contributions:

- Wang and Appel's collector [WA01] relies on whole-program analysis and code duplication to support higher-order and polymorphic languages; this breaks separate compilation. We show how to use runtime type analysis to write our GC as a library (thus no code duplication) and how to directly support higher-order polymorphic functions.
- Monomorphization is not applicable in the presence of recursive polymorphism or existential packages, so their type-safe GC cannot handle languages with polymorphic recursion or abstract types. Our system does not have this disadvantage.

4.1.1 The problem

A region calculus [TT94] annotates the type of every heap object with the region in which it is allocated. For example, a pair of values will have the type $\sigma_1 \times^\rho \sigma_2$, where ρ is the region in which the pair is allocated. Thus the type of all the live values reflect all the live regions; any region that does not appear in any of the currently live types can be safely reclaimed.

In a type-safe copying GC, we make no correctness guarantees. Suppose that the collector gets invoked when a region F gets filled. Rather than prove that the *copy* function faithfully copies the heap to a new region T, we simply show that it has the type $\forall \alpha.(\alpha \rightarrow ([\mathsf{T}/\mathsf{F}]\alpha))$ where $([\mathsf{T}/\mathsf{F}]\alpha)$ denotes the type α with the region annotation T substituted for F (see Fig. 4.1). The region calculus will then allow us to safely reclaim F.

The main problem is to write this *copy* function in a way that allows it to trace through arbitrary heap structures at runtime, and to design a type system that is sophisticated enough to express its type.

4.1.2 Our solution

The substitution present in the return type of *copy* as well as the need to observe types at runtime leads one very naturally to the framework presented in Chapter 3. The substitution can be expressed in a straightforward way by using the Typerec construct, while we can use the typecase construct to inspect types at runtime. In fact, there are only a few remaining issues involved in getting the right framework.

Consider again the type $\forall\alpha.(\alpha \rightarrow ([\mathsf{T}/\mathsf{F}]\alpha))$ of the *copy* function. The type of an object grows every time it is copied. If the object had the type σ to begin with, then the type changes to $[\mathsf{T}/\mathsf{F}]\sigma$, and then to $[\mathsf{T}'/\mathsf{T}]([\mathsf{T}/\mathsf{F}]\sigma)$ (where T' is the new region after the second collection), ... Since σ may contain type variables, the substitution may not get reduced away: $[\mathsf{T}/\mathsf{F}]\alpha$ cannot be reduced further until α is instantiated. In other words, a type such as $\exists\alpha.[\mathsf{T}/\mathsf{F}]\alpha$ is a normal form. This causes a problem since $[\mathsf{T}'/\mathsf{F}]\alpha$ is not equal to $[\mathsf{T}'/\mathsf{T}]([\mathsf{T}/\mathsf{F}]\alpha)$. Thus we must ensure that the input and output types of *copy* are symmetric. We first define $\mathsf{S}_\rho(\sigma)$ as substituting ρ for any region annotation and then redefine *copy* to have type $\forall\mathsf{F}.\forall\mathsf{T}.\forall\alpha.(\mathsf{S}_\mathsf{F}(\alpha) \rightarrow \mathsf{S}_\mathsf{T}(\alpha))$. This ensures that *GC* does not increase the size of the type anymore, and also gets rid of the special case before the first collection.

The above solution looks good until we try to copy existential packages $\exists\alpha\in\Theta.\sigma$, that are used for encoding closures. The Θ annotation bounds the set of regions that can appear in the witness type hidden under the type variable α. Opening an existential package of type $\exists\alpha\in\overline{\mathsf{F}}.\mathsf{S}_\mathsf{F}(\alpha)$, gives us the witness type σ, and a value of type $\mathsf{S}_\mathsf{F}(\sigma)$. After getting copied this package should have the type $\exists\alpha \in \overline{\mathsf{T}}.\mathsf{S}_\mathsf{T}(\alpha)$. Recursively applying *copy* to the value will return a new value of type $\mathsf{S}_\mathsf{T}(\sigma)$, but what happens to the witness type? Reusing σ for the witness type will not do since σ is not constrained to $\overline{\mathsf{T}}$ but to $\overline{\mathsf{F}}$. A witness of $\mathsf{S}_\mathsf{T}(\sigma)$ does not work either; the only correctly typed package we can produce then is $\langle\alpha=\mathsf{S}_\mathsf{T}(\sigma), v : \alpha\rangle$ which has type $\exists\alpha\in\overline{\mathsf{T}}.\alpha$.

The problem arises because, on the one hand, an existential type reveals no information about the witness type; on the other, we need to suitably constrain the region annotation for the witness type. We will get around this problem by defining a parallel set of non-annotated types τ (that

we will call *tags*). The witness for an existential type will now be a tag. Note that contrary to common practice, our tags are not attached to their corresponding objects but are managed completely independently.

Such a split between types and tags is reminiscent of the type system for predicative languages where tags were called *constructors* [HM95, CW99]. But, in these languages, the projection from tags to types is essentially an identity (Section 3.2.1). Here tags take on more significance and will be mapped to actual types via type-level operators that enhance the tags with a lot more information. In essence, this information encapsulates the constraints that mutator data has to satisfy in order for the collector to do its job.

4.2 Source language λ_{CLOS}

For simplicity of the presentation, the source language we propose to compile and garbage collect is the simply typed λ-calculus. In order to be able to use our region calculus, we need to convert the source program into a continuation passing style form (CPS). We also need to close our code to make all data manipulation explicit, so we turn all closures into existential packages. We will not go into the details of how to do the CPS conversion [DF92] and the closure conversion [MMH96, HM98] since the algorithms are well known.

The language used after CPS conversion and closure conversion is the language λ_{CLOS} shown below.

$$
\begin{array}{lll}
(\textit{types}) & \tau ::= \mathsf{Int} \mid \alpha \mid \tau_1 \times \tau_2 \mid \tau \rightarrow 0 \mid \exists \alpha.\tau \\
(\textit{values}) & v ::= n \mid f \mid x \mid (v_1, v_2) \mid \langle \alpha = \tau_1, v : \tau_2 \rangle \\
(\textit{terms}) & e ::= \mathsf{let}\ x = v\ \mathsf{in}\ e \mid \mathsf{let}\ x = \pi_i v\ \mathsf{in}\ e \mid v_1(v_2) \\
 & \quad \mid \mathsf{open}\ v\ \mathsf{as}\ \langle \alpha, x \rangle\ \mathsf{in}\ e \mid \mathsf{halt}\ v \\
(\textit{programs}) & p ::= \mathsf{letrec}\ \overrightarrow{f = \lambda(x : \tau).e}\ \mathsf{in}\ e
\end{array}
$$

Since functions are in CPS, they never return, which we represent with the arbitrary return type 0, often referred to as *void*. The construct (v_1, v_2) represents a pair while $\pi_i v$ selects its i^{th} element. To represent closures, the language includes existential packages $\langle \alpha = \tau_1, v : \tau_2 \rangle$ of type $\exists \alpha.\tau_2$. The abstract type α hides the witness type τ_1. Therefore, the value v has the actual type $[\tau_1/\alpha]\tau_2$. The construct $\mathsf{open}\ v\ \mathsf{as}\ \langle \alpha, x \rangle\ \mathsf{in}\ e$ takes an existential package v, binds the witness type to α and the value to x, and then executes e. The complete program $\mathsf{letrec}\ \overrightarrow{f_i = \lambda(x_i : \tau_i).e_i}\ \mathsf{in}\ e$

(*regions*)	ρ	$::= \nu \mid r$
(*kinds*)	κ	$::= \Omega \mid \Omega \rightarrow \Omega$
(*tags*)	τ	$::= \alpha \mid \mathsf{Int} \mid \tau_1 \times \tau_2 \mid \tau \rightarrow 0 \mid \exists \alpha.\tau$
		$\mid \lambda \alpha\!:\!\kappa.\, \tau \mid \tau_1\, \tau_2$
(*types*)	σ	$::= \mathsf{int} \mid \sigma_1 \times \sigma_2 \mid \forall [\alpha \vec{:} \kappa][\vec{r}](\vec{\sigma}) \rightarrow 0 \mid \exists \alpha : \kappa.\sigma \mid \sigma\ \mathsf{at}\ \rho$
		$\mid \mathsf{M}_\rho(\tau)$
(*values*)	v	$::= n \mid x \mid \nu.\ell \mid (v_1, v_2) \mid \langle \alpha = \tau, v : \sigma \rangle \mid \lambda [\alpha \vec{:} \kappa][\vec{r}](\overrightarrow{x : \sigma}).e$
(*operations*)	op	$::= v \mid \pi_i v \mid \mathsf{put}[\rho] v \mid \mathsf{get}\ v$
(*terms*)	e	$::= v[\vec{\tau}][\vec{\rho}](\vec{v}) \mid \mathsf{let}\ x = op\ \mathsf{in}\ e \mid \mathsf{halt}\ v \mid \mathsf{ifgc}\ \rho\ e_1\ e_2$
		$\mid \mathsf{open}\ v\ \mathsf{as}\ \langle \alpha, x \rangle\ \mathsf{in}\ e \mid \mathsf{let\ region}\ r\ \mathsf{in}\ e \mid \mathsf{only}\ \Theta\ \mathsf{in}\ e$
		$\mid \mathsf{typecase}\ \tau\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; \alpha_1 \alpha_2 . e_{\times}; \alpha_e . e_{\exists})$
(*normal tags*)	τ'	$::= \alpha \mid \mathsf{Int} \mid \tau' \rightarrow 0 \mid \tau'_1 \times \tau'_2 \mid \exists \alpha.\tau' \mid \lambda \alpha\!:\!\kappa.\, \tau' \mid \alpha\, \tau'$

Figure 4.2: Syntax of λ_{GC}

consists of a list of mutually recursive closed function declarations followed by the main term to be executed.

4.3 Target language λ_{GC}

We translate λ_{CLOS} programs into our target language λ_{GC}. The target language is also used to write the garbage collector. λ_{GC} extends λ_{CLOS} with regions [TT94] and fully reflexive intensional type analysis. The syntax of $\lambda_{\mathsf{GC}}$is shown in Figure 4.2. The static semantics is shown in Figures 4.3 through 4.5.

4.3.1 Functions and code

Since programs in λ_{GC} are completely closed, we can separate code from data. The memory configuration enforces this by having a separate dedicated region cd for all the code blocks. A value $\lambda [\alpha \vec{:} \kappa][\vec{r}](\overrightarrow{x : \sigma}).e$ is only an array of instructions (which can contain references to other values in cd) and needs to be put into a region to get a function pointer before one can call it. In practice, functions are placed into the cd region when translating code from λ_{CLOS} and never directly appear in λ_{GC} code.

The indirection provided by memory references allows us to do away with letrec. For convenience, we will use $\mathsf{fix} f.e$ in examples, but in reality, e will be placed at the location ℓ in the cd region and all occurrences of f will be replaced by $\mathsf{cd}.\ell$. We treat cd as a special region. It cannot

be freed and can only contain functions, no other kind of data.

4.3.2 The type calculus

Since the garbage collector needs to know the type of values at runtime, the language λ_{GC} must support the runtime analysis of types. Therefore, conceptually, types need to play a dual role in this language. As in the source language λ_{CLOS}, they are used at compile time to type-check well formed terms. However, they are also used at runtime, as tags, to be inspected by the garbage collector. To enforce this distinction, we split types into a tag language and a type language. The tags correspond to the runtime entity, while the types correspond to the compile time entity.

The tag for a value is constructed during the translation from λ_{CLOS} to λ_{GC}. In fact, the tags closely resemble the λ_{CLOS} types. We only need to add tag-level functions ($\lambda\alpha\!:\!\kappa.\,\tau$) and tag-level applications ($\tau\,\tau_1$) to support tag analysis. In turn, this requires adding the function kind $\Omega \rightarrow \Omega$. The type-level analysis is done by the M operator that is defined using a Typerec. As before (Section 3.3.1), we will use ML-style pattern matching to define this type:

$$
\begin{aligned}
\mathsf{M}_\rho(\mathsf{Int}) &= \mathsf{int} \\
\mathsf{M}_\rho(\tau_1 \times \tau_2) &= (\mathsf{M}_\rho(\tau_1) \times \mathsf{M}_\rho(\tau_2))\ \mathsf{at}\ \rho \\
\mathsf{M}_\rho(\exists\alpha.\tau) &= (\exists\alpha : \Omega.\mathsf{M}_\rho(\tau))\ \mathsf{at}\ \rho \\
\mathsf{M}_\rho(\tau \rightarrow 0) &= \forall[][r](\mathsf{M}_r(\tau)) \rightarrow 0\ \mathsf{at}\ \mathsf{cd}
\end{aligned}
$$

$\mathsf{M}_\rho(\tau)$ is the type corresponding to the tag τ augmented with region annotations ρ. The definition of M captures the invariant that all objects are allocated in the same region.

Types are used to classify terms. The type language includes the existential type for typing closures and the code type $\forall[\vec{\alpha}][\vec{r}](\vec{\sigma}) \rightarrow 0$ for fully closed CPS functions. Moreover, types in the target language must include the region in which the corresponding value resides. Therefore, we use the notation $\sigma\ \mathsf{at}\ \rho$ for the type of a value of type σ in region ρ.

4.3.3 The term calculus

The term language must support region based memory management and runtime type analysis. New regions are created through the $\mathsf{let\ region}\ r\ \mathsf{in}\ e$ construct which allocates a new region ν at runtime and binds r to it. A term of the form $\mathsf{put}[\rho]v$ allocates a value v in the region ρ. Data is read from a region in two ways. Functions are read implicitly through a function call. Data may also be read through the $\mathsf{get}\ v$ construct.

$\boxed{\Delta \vdash \tau \ : \ \kappa}$

$$\frac{}{\cdot \vdash \mathsf{Int} \ : \ \Omega} \qquad \frac{\Delta(\alpha) = \kappa}{\Delta \vdash \alpha \ : \ \kappa} \qquad \frac{\Delta \vdash \tau_1 \ : \ \Omega \quad \Delta \vdash \tau_2 \ : \ \Omega}{\Delta \vdash \tau_1 \times \tau_2 \ : \ \Omega}$$

$$\frac{\Delta \vdash \tau_i \ : \ \Omega}{\Delta \vdash \vec{\tau} \to 0 \ : \ \Omega} \qquad \frac{\Delta, \alpha{:}\Omega \vdash \tau \ : \ \Omega}{\Delta \vdash \exists \alpha.\tau \ : \ \Omega}$$

$$\frac{\Delta, \alpha{:}\Omega \vdash \tau \ : \ \Omega}{\Delta \vdash \lambda \alpha{:}\Omega.\, \tau \ : \ \Omega \to \Omega} \qquad \frac{\Delta \vdash \tau_1 \ : \ \Omega \to \Omega \qquad \Delta \vdash \tau_2 \ : \ \Omega}{\Delta \vdash \tau_1 \, \tau_2 \ : \ \Omega}$$

$\boxed{\Theta; \Delta \vdash \sigma}$

$$\frac{}{\Theta; \Delta \vdash \mathsf{int}} \qquad \frac{\Theta; \Delta \vdash \sigma_1 \qquad \Theta; \Delta \vdash \sigma_2}{\Theta; \Delta \vdash \sigma_1 \times \sigma_2}$$

$$\frac{\overline{\vec{r}}; \alpha \, \vec{:} \, \kappa \vdash \sigma_i}{\Theta; \Delta \vdash \forall [\alpha \, \vec{:} \, \kappa][\vec{r}](\vec{\sigma}) \to 0} \qquad \frac{\Theta; \Delta, \alpha{:}\kappa \vdash \sigma}{\Theta; \Delta \vdash \exists \alpha : \kappa.\sigma}$$

$$\frac{\Theta; \Delta \vdash \sigma \qquad \rho \in \Theta}{\Theta; \Delta \vdash \sigma \ \mathsf{at} \ \rho} \qquad \frac{\Delta \vdash \tau \ : \ \Omega \qquad \rho \in \Theta}{\Theta; \Delta \vdash \mathsf{M}_\rho(\tau)}$$

Figure 4.3: Type and tag formation rules

Operationally, get takes a memory address $\nu.\ell$ and dereferences it. Since our region calculus does not admit dangling references, and since each reference implicitly carries a region handle, get does not need a region argument, as opposed to put.

Deallocation is handled implicitly through the only Θ in e construct [WA99]. It asserts statically that the expression e can be evaluated using only the set of regions in Θ' (i.e. Θ extended with the cd region), which is a subset of the regions currently in scope. At runtime, an implementation would treat the set of regions in Θ' as live and reclaim all other regions.

$$\frac{\Theta' = \Theta, \mathsf{cd} \quad \Psi|_{\Theta'}; \Theta'; \Delta; \Gamma|_{\Theta'} \vdash e \quad \Theta' \subset \Theta''}{\Psi; \Theta''; \Delta; \Gamma \vdash \mathsf{only}\ \Theta\ \mathsf{in}\ e}$$

The construct $|_{\Theta'}$ restricts an environment to the set of regions in Θ', i.e. $\Psi|_{\Theta'}$ is the subset of the heap restricted to the regions in Θ'. Similarly, $\Gamma|_{\Theta'}$ eliminates from Γ all variables whose type refers to regions not mentioned in Θ'.

The use of only was chosen for its simplicity. Other approaches either do not work with a CPS language or carry a significant added complexity to handle the problem of aliasing. The only construct side steps this difficulty by making the actual deletion implicit: instead of explicitly requesting the deletion of a region (say r_1), the program will request to keep a region (say r_2). At runtime the system checks to see that r_1 is not aliased to r_2 and only then deletes it. In order to trigger GC, ifgc allows us to check whether a region is full. We will not consider the exact mechanism for implementing this check.

The runtime type analysis is handled through a typecase construct. Depending on the head of the tag being analyzed, typecase chooses one of the branches for execution. When analyzing a tag variable α, we refine types containing α in each of the branches [CWM98].

$$\frac{\begin{array}{l}\Delta \vdash \alpha\ :\ \Omega \\ \Psi; \Theta; \Delta; [\mathsf{int}/\alpha]\Gamma \vdash [\mathsf{int}/\alpha]e_i \\ \ldots\end{array}}{\Psi; \Theta; \Delta; \Gamma \vdash \mathsf{typecase}\ \alpha\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; \alpha_1\alpha_2.e_{\times}; \alpha_e.e_{\exists})}$$

In the e_i branch, we know that the tag variable α is bound to Int and can therefore substitute it away. A similar rule is applied to the other cases.

Programs in λ_{GC} use an allocation semantics which makes the allocation of data in memory

$\boxed{\Psi;\Theta;\Delta;\Gamma \vdash v \ : \ \sigma \qquad \Psi;\Theta;\Delta;\Gamma \vdash op \ : \ \sigma}$

$$\frac{}{\Psi;\Theta;\Delta;\Gamma \vdash n \ : \ \mathsf{int}} \qquad \frac{\Gamma(x) = \sigma}{\Psi;\Theta;\Delta;\Gamma \vdash x \ : \ \sigma}$$

$$\frac{\Psi(\nu.\ell) = \sigma \quad Dom(\Psi);\cdot \vdash \sigma \ \mathsf{at} \ \nu}{\Psi;\Theta;\Delta;\Gamma \vdash \nu.\ell \ : \ \sigma \ \mathsf{at} \ \nu}$$

$$\frac{\mathsf{cd}, \vec{r}; \alpha \vec{:} \kappa \vdash \sigma_i \qquad \Psi|_{\mathsf{cd}}; \mathsf{cd}, \vec{r}; \overrightarrow{\alpha : \kappa}; \overrightarrow{x : \sigma} \vdash e}{\Psi;\Theta;\Delta;\Gamma \vdash \lambda[\alpha \vec{:} \kappa][\vec{r}](\overrightarrow{x : \sigma}).e \ : \ \forall[\alpha \vec{:} \kappa][\vec{r}](\vec{\sigma}) \to 0}$$

$$\frac{\Psi;\Theta;\Delta;\Gamma \vdash v_1 \ : \ \sigma_1 \qquad \Psi;\Theta;\Delta;\Gamma \vdash v_2 \ : \ \sigma_2}{\Psi;\Theta;\Delta;\Gamma \vdash (v_1, v_2) \ : \ \sigma_1 \times \sigma_2}$$

$$\frac{\Psi;\Theta;\Delta;\Gamma \vdash v \ : \ \sigma_1 \times \sigma_2}{\Psi;\Theta;\Delta;\Gamma \vdash \pi_i v \ : \ \sigma_i} \qquad \frac{\Psi;\Theta;\Delta;\Gamma \vdash v \ : \ \sigma \ \mathsf{at} \ \rho}{\Psi;\Theta;\Delta;\Gamma \vdash \mathsf{get} \ v \ : \ \sigma}$$

$$\frac{\Delta \vdash \tau \ : \ \kappa \quad \Psi;\Theta;\Delta;\Gamma \vdash v \ : \ [\tau/\alpha]\sigma}{\Psi;\Theta;\Delta;\Gamma \vdash \langle \alpha = \tau, v : \sigma \rangle \ : \ \exists \alpha : \kappa.\sigma}$$

$$\frac{\Psi;\Theta;\Delta;\Gamma \vdash v \ : \ \sigma \qquad \rho \in \Theta}{\Psi;\Theta;\Delta;\Gamma \vdash \mathsf{put}[\rho] v \ : \ \sigma \ \mathsf{at} \ \rho}$$

Figure 4.4: Formation rules for λ_{GC} values

$\boxed{\Psi;\Theta;\Delta;\Gamma \vdash e}$

$$\frac{\Psi;\Theta;\Delta;\Gamma \vdash v \ : \ \forall[\alpha \vec{:} \kappa][\vec{r}](\vec{\sigma}) \to 0 \text{ at } \rho \quad \Psi;\Theta;\Delta;\Gamma \vdash v_i \ : \ [\vec{\rho},\vec{\tau}/\vec{r},\vec{\alpha}]\sigma_i \qquad \Delta \vdash \tau_i \ : \ \kappa_i \qquad \rho_i \in \Theta}{\Psi;\Theta;\Delta;\Gamma \vdash v[\vec{\tau}][\vec{\rho}](\vec{v})}$$

$$\frac{\Psi;\Theta;\Delta;\Gamma \vdash op \ : \ \sigma \qquad \Psi;\Theta;\Delta;\Gamma, x:\sigma \vdash e}{\Psi;\Theta;\Delta;\Gamma \vdash \mathsf{let}\ x = op\ \mathsf{in}\ e}$$

$$\frac{\Psi;\Theta;\Delta;\Gamma \vdash v \ : \ \exists\alpha' : \kappa.\sigma \quad \Psi;\Theta;\Delta,\alpha:\kappa;\Gamma, x : [\alpha/\alpha']\sigma \vdash e}{\Psi;\Theta;\Delta;\Gamma \vdash \mathsf{open}\ v\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ e}$$

$$\frac{\Psi;\Theta;\Delta;\Gamma \vdash e_1 \quad \Psi;\Theta;\Delta;\Gamma \vdash e_2 \quad \rho \in \Theta}{\Psi;\Theta;\Delta;\Gamma \vdash \mathsf{ifgc}\ \rho\ e_1\ e_2}$$

$$\frac{\Psi;\Theta, r;\Delta;\Gamma \vdash e}{\Psi;\Theta;\Delta;\Gamma \vdash \mathsf{let\ region}\ r\ \mathsf{in}\ e} \qquad \frac{\Psi;\Theta;\Delta;\Gamma \vdash v \ : \ \mathsf{int}}{\Psi;\Theta;\Delta;\Gamma \vdash \mathsf{halt}\ v}$$

$$\frac{\Psi|_{\Theta'};\Theta', \mathsf{cd};\Delta|_{\Theta'};\Gamma|_{\Theta'} \vdash e \quad \Theta' \subset \Theta}{\Psi;\Theta;\Delta;\Gamma \vdash \mathsf{only}\ \Theta'\ \mathsf{in}\ e}$$

$$\frac{\Psi;\Theta;\Delta;\Gamma \vdash e_{\mathsf{i}}}{\Psi;\Theta;\Delta;\Gamma \vdash \mathsf{typecase\ Int\ of}\ (e_{\mathsf{i}}; e_{\to}; \alpha_1\alpha_2.e_{\times}; \alpha_e.e_{\exists})}$$

$$\frac{\Psi;\Theta;\Delta;\Gamma \vdash e_{\to}}{\Psi;\Theta;\Delta;\Gamma \vdash \mathsf{typecase}\ \vec{\tau} \to 0\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\to}; \alpha_1\alpha_2.e_{\times}; \alpha_e.e_{\exists})}$$

$$\frac{\Psi;\Theta;\Delta;\Gamma \vdash [\tau_1, \tau_2/\alpha_1, \alpha_2]e_{\times}}{\Psi;\Theta;\Delta;\Gamma \vdash \mathsf{typecase}\ (\tau_1 \times \tau_2)\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\to}; \alpha_1\alpha_2.e_{\times}; \alpha_e.e_{\exists})}$$

$$\frac{\Psi;\Theta;\Delta;\Gamma \vdash [\lambda\alpha{:}\Omega.\,\tau/\alpha_e]e_{\exists}}{\Psi;\Theta;\Delta;\Gamma \vdash \mathsf{typecase}\ \exists\alpha.\tau\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\to}; \alpha_1\alpha_2.e_{\times}; \alpha_e.e_{\exists})}$$

$$\frac{\begin{array}{l}\Delta \vdash \alpha \ : \ \Omega \\ \Psi;\Theta;\Delta;[\mathsf{int}/\alpha]\Gamma \vdash [\mathsf{int}/\alpha]e_{\mathsf{i}} \\ \Psi;\Theta;\Delta;\Gamma \vdash e_{\to} \\ \Psi;\Theta;\Delta,\alpha_1:\Omega,\alpha_2:\Omega;[\alpha_1\times\alpha_2/\alpha]\Gamma \vdash [\alpha_1\times\alpha_2/\alpha]e_{\times} \\ \Psi;\Theta;\Delta,\alpha_e:\Omega\to\Omega;[\exists\alpha'.\alpha_e\,\alpha'/\alpha]\Gamma \vdash [\exists\alpha'.\alpha_e\,\alpha'/\alpha]e_{\exists}\end{array}}{\Psi;\Theta;\Delta;\Gamma \vdash \mathsf{typecase}\ \alpha\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\to}; \alpha_1\alpha_2.e_{\times}; \alpha_e.e_{\exists})}$$

Figure 4.5: Term formation rules of λ_{GC}.

$(M, \nu.\ell[\vec{\tau}][\vec{\rho}](\vec{v}))$	$\leadsto (M, \nu.\ell[\vec{\tau'}][\vec{\rho}](\vec{v}))$
$(M, \nu.\ell[\vec{\tau'}][\vec{\rho}](\vec{v}))$ where $M(\nu.\ell) = (\lambda[\alpha \vec{:} \kappa][\vec{r}](\vec{x} : \vec{\sigma}).e)$	$\leadsto (M, e[\vec{\rho}, \vec{\tau'}, \vec{v}/\vec{r}, \vec{\alpha}, \vec{x}])$
$(M, \mathsf{let}\ x = v\ \mathsf{in}\ e)$	$\leadsto (M, e[v/x])$
$(M, \mathsf{let}\ x = \pi_i(v_1, v_2)\ \mathsf{in}\ e)$	$\leadsto (M, e[v_i/x])$
$(M, \mathsf{let}\ x = \mathsf{put}[\nu]v\ \mathsf{in}\ e)$ where $\ell \notin Dom(M(\nu))$	$\leadsto (M\{\nu.\ell \mapsto v\}, [\nu.\ell/x]e)$
$(M, \mathsf{let}\ x = \mathsf{get}\ \nu.\ell\ \mathsf{in}\ e)$	$\leadsto (M, [v/x]e)$ where $M(\nu.\ell) = v$
$(M, \mathsf{open}\ \langle \alpha{=}\tau, v : \sigma\rangle\ \mathsf{as}\ \langle \alpha, x\rangle\ \mathsf{in}\ e)$	$\leadsto (M, \mathsf{open}\ \langle \alpha{=}\tau', v : \sigma\rangle\ \mathsf{as}\ \langle \alpha, x\rangle\ \mathsf{in}\ e)$
$(M, \mathsf{open}\ \langle \alpha{=}\tau', v : \sigma\rangle\ \mathsf{as}\ \langle \alpha, x\rangle\ \mathsf{in}\ e)$	$\leadsto (M, e[\tau', v/\alpha, x])$
$(M, \mathsf{ifgc}\ \rho\ e_1\ e_2)$	$\leadsto (M, e_1)$ if ρ is full
$(M, \mathsf{ifgc}\ \rho\ e_1\ e_2)$	$\leadsto (M, e_2)$ if ρ is not full
$(M, \mathsf{let\ region}\ r\ \mathsf{in}\ e)$ where $\nu \notin Dom(M)$	$\leadsto (M\{\nu \mapsto \{\}\}, e[\nu/r])$
$(M, \mathsf{only}\ \Theta\ \mathsf{in}\ e)$	$\leadsto (M|_\Theta, e)$
$(M, \mathsf{typecase}\ \tau\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; \alpha_1\alpha_2.e_{\times}; \alpha_e.e_{\exists}))$	$\leadsto (M, \mathsf{typecase}\ \tau'\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; \alpha_1\alpha_2.e_{\times}; \alpha_e.e_{\exists}))$
$(M, \mathsf{typecase}\ \mathsf{Int}\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; \alpha_1\alpha_2.e_{\times}; \alpha_e.e_{\exists}))$	$\leadsto (M, e_i)$
$(M, \mathsf{typecase}\ \tau \rightarrow 0\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; \alpha_1\alpha_2.e_{\times}; \alpha_e.e_{\exists}))$	$\leadsto (M, e_{\rightarrow})$
$(M, \mathsf{typecase}\ \tau_1 \times \tau_2\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; \alpha_1\alpha_2.e_{\times}; \alpha_e.e_{\exists}))$	$\leadsto (M, [\tau_1, \tau_2/\alpha_1, \alpha_2]e_{\times})$
$(M, \mathsf{typecase}\ \exists\alpha.\tau\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; \alpha_1\alpha_2.e_{\times}; \alpha_e.e_{\exists}))$	$\leadsto (M, [\lambda\alpha{:}\Omega.\,\tau/\alpha_e]e_{\exists})$

Figure 4.6: Operational semantics of λ_{GC}

explicit. The operational semantics, defined in Fig. 4.6, maps a machine state P to a new machine state P'. A machine state is a pair (M, e) of a memory M and a term e being executed. A memory consists of a set of regions; hence, it is defined formally as a map between region names ν and regions R. A region, in turn, is a map from offsets ℓ to storable values v. Therefore, an address is given by a pair of a region and an offset $\nu.\ell$. We assign a type to every location allocated in a region with the memory environment Ψ. Figure 4.7 shows the form of environments.

The language λ_{GC} obeys the following properties. The proofs are given in Appendix B.

Proposition 4.3.1 (Type Preservation) *If* $\vdash (M, e)$ *and* $(M, e) \leadsto (M', e')$ *then* $\vdash (M', e')$.

Proposition 4.3.2 (Progress) *If* $\vdash (M, e)$ *then either* $e = \mathsf{halt}\ v$ *or there exists a* (M', e') *such that* $(M, e) \leadsto (M', e')$.

$\boxed{\Theta \vdash \Upsilon \qquad \vdash \Psi}$

$$\frac{\Theta; \cdot \vdash \sigma_i}{\Theta \vdash \overline{\ell_1 : \sigma_1, \ldots, \ell_n : \sigma_n}} \qquad \frac{\overline{\nu_1, \ldots, \nu_n} \vdash \Upsilon_i}{\vdash \overline{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n}}$$

$$\Upsilon_{\mathsf{cd}} = \overline{\ell_1 : \forall[\vec{\tau_1}][\vec{r_1}](\overrightarrow{v_1 : \sigma_1}) \to 0, \ldots, \ell_n : \forall[\vec{\tau_n}][\vec{r_n}](\overrightarrow{v_n : \sigma_n}) \to 0}$$

$\boxed{\Psi \vdash R \ : \ \Upsilon \qquad \vdash M \ : \ \Psi}$

$$\frac{\Psi; Dom(\Psi); \cdot; \cdot \vdash v_i \ : \ \sigma_i}{\Psi \vdash \overline{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n} \ : \ \overline{\ell_1 : \sigma_1, \ldots, \ell_n : \sigma_n}}$$

$$\frac{\vdash \overline{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n} \qquad \overline{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n} \vdash R_i \ : \ \Upsilon_i}{\vdash \overline{\nu_1 \mapsto R_1, \ldots, \nu_n \mapsto R_n} \ : \ \overline{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n}}$$

Figure 4.7: Environment formation rules.

4.4 Translating λ_{CLOS} to λ_{GC}

The translation of terms from λ_{CLOS} to λ_{GC} (Fig. 4.8) is mostly directed by the type translation M_ρ presented earlier: each function takes the current region as an argument and begins by checking if a garbage collection is necessary. All operations on data are slightly rewritten to account for the need to allocate them in the region or to fetch them from the region. For example a λ_{CLOS} function like:

$$\begin{array}{l}\mathsf{fix}\ swap(x : \mathsf{Int} \times \mathsf{Int}).\\ \quad \mathsf{let}\ x1 = \pi_1 x\ \mathsf{in}\ \mathsf{let}\ x2 = \pi_2 x\ \mathsf{in}\ \mathsf{let}\ x' = (x2, x1)\ \mathsf{in}\ \mathsf{halt}\ 0\end{array}$$

will be turned into the following λ_{GC} function:

$$\begin{array}{l}\mathsf{cd}.\ell = \lambda[][r](x : (\mathsf{int} \times \mathsf{int})\ \mathsf{at}\ r).\\ \qquad \mathsf{ifgc}\ r\ (gc[\mathsf{Int} \times \mathsf{Int}][r](\mathsf{cd}.\ell, x))\\ \qquad\quad \mathsf{let}\ x = \mathsf{get}\ x\ \mathsf{in}\\ \qquad\quad \mathsf{let}\ x1 = \pi_1 x\ \mathsf{in}\\ \qquad\quad \mathsf{let}\ x2 = \pi_2 x\ \mathsf{in}\\ \qquad\quad \mathsf{let}\ x' = \mathsf{put}[r](x2, x1)\ \mathsf{in}\\ \qquad\quad \mathsf{halt}\ 0\end{array}$$

The mapping between λ_{CLOS} identifiers like $swap$ and λ_{GC} location like $\mathsf{cd}.\ell$ is kept in the

$\boxed{F \vdash \lambda_{\mathsf{CLOS}} \Rightarrow \lambda_{\mathsf{GC}}}$

$$\overline{F \vdash_v n \Rightarrow n} \qquad \overline{F \vdash_v f \Rightarrow \mathsf{cd}.F(f)} \qquad \overline{F \vdash_v x \Rightarrow x}$$

$$\frac{F \vdash_v v_1 \Rightarrow v_1' \quad F \vdash_v v_2 \Rightarrow v_2'}{F \vdash_v (v_1, v_2) \Rightarrow \mathsf{put}[r](v_1', v_2')}$$

$$\frac{F \vdash_v v \Rightarrow v'}{F \vdash_v \langle \alpha{=}\tau_1, v : \tau_2 \rangle \Rightarrow \mathsf{put}[r]\langle \alpha{=}\tau_1, v' : \mathsf{M}_r(\tau_2) \rangle}$$

$$\frac{F \vdash_v v_1 \Rightarrow v_1' \quad F \vdash_v v_2 \Rightarrow v_2'}{F \vdash_e v_1(v_2) \Rightarrow v_1'[][r](v_2')} \qquad \frac{F \vdash_v v \Rightarrow v'}{F \vdash_e \mathsf{halt}\ v \Rightarrow \mathsf{halt}\ v'}$$

$$\frac{F \vdash_e e \Rightarrow e' \quad F \vdash_v v \Rightarrow v'}{F \vdash_e \mathsf{open}\ v\ \mathsf{as}\ \langle \alpha, x \rangle\ \mathsf{in}\ e \Rightarrow \mathsf{open}\ (\mathsf{get}\ v')\ \mathsf{as}\ \langle \alpha, x \rangle\ \mathsf{in}\ e'}$$

$$\frac{F \vdash_e e \Rightarrow e' \quad F \vdash_v v \Rightarrow v'}{F \vdash_e \mathsf{let}\ x = v\ \mathsf{in}\ e \Rightarrow \mathsf{let}\ x = v'\ \mathsf{in}\ e'}$$

$$\frac{F \vdash_e e \Rightarrow e' \quad F \vdash_v v \Rightarrow v'}{F \vdash_e \mathsf{let}\ x = \pi_i v\ \mathsf{in}\ e \Rightarrow \mathsf{let}\ x = \pi_i(\mathsf{get}\ v')\ \mathsf{in}\ e'}$$

$$\frac{F \vdash_e e \Rightarrow e' \quad \ell = F(f)}{\begin{array}{l} \vdash_f (f = \lambda(x : \tau).e) \\ \quad \Rightarrow\ \lambda[][r](x : \mathsf{M}_r(\tau)).\mathsf{ifgc}\ r\ (gc[\tau][r](\mathsf{cd}.\ell, x))\ e' \end{array}}$$

$$\frac{\begin{array}{c} F = \overline{f_1 \mapsto \ell_1, \ldots, f_n \mapsto \ell_n} \\ F \vdash_f f_i = \lambda(x_i : \tau_i).e_i \Rightarrow f_i' \qquad F \vdash_e e \Rightarrow e' \end{array}}{\begin{array}{l} \vdash_p \mathsf{letrec}\ \overrightarrow{f_i = \lambda(x_i : \tau_i).e_i}\ \mathsf{in}\ e \\ \quad \Rightarrow\ (\mathsf{cd} \mapsto \overline{\ell_1 \mapsto f_1', \ldots}, \mathsf{let\ region}\ r\ \mathsf{in}\ e') \end{array}}$$

Figure 4.8: Translation of λ_{CLOS} terms.

```
fix gc[α : Ω][r1](f : ∀[][r](M_r(α)) → 0, x : M_{r1}(α)).
    let region r2 in
    let y = copy[α][r1, r2](x) in
    only r̄2 in f[][r2](y)

fix copy[α : Ω][r1, r2](x : M_{r1}(α)) : M_{r2}(α).
    typecase α of
        Int     ⇒ x
        →       ⇒ x
      α1 × α2   ⇒ let x1 = copy[α1][r1, r2](π1(get x)) in
                  let x2 = copy[α2][r1, r2](π2(get x)) in
                  put[r2](x1, x2)
      ∃αe       ⇒ open (get x) as ⟨α, y⟩ in
                  let z = copy[αe α][r1, r2](y) in
                  put[r2]⟨α=α, z : M_{r2}(αe α)⟩
```

Figure 4.9: The garbage collector proper.

environment F. The region argument r refers to the current region. It is initially created at the very beginning of the program and is changed after each garbage collection.

An important detail here is that the garbage collector receives the tag τ rather than the type σ of the argument. The GC receives the tags for analysis as they were in λ_{CLOS} rather than as they are translated in λ_{GC}.

To make the presentation simpler, the garbage-collection code in Fig. 4.9 uses some syntactic sugar and resorts to a direct-style presentation of the *copy* function. In [MSS00] we show the same code after CPS and closure conversion. The conversion does not present any new technical difficulties, but just makes the code a lot harder to understand. We will therefore stick to the direct style presentation here. The garbage collector itself is very simple: it first allocates the *to* region, asks *copy* to move everything into it and then frees the *from* region before jumping to its continuation (which uses the new region).

The *copy* function is similarly straightforward, recursing over the whole heap and copying in a depth-first way. The direct style here hides the stack. When the code is CPS converted and closed, we have to allocate that stack of continuations in an additional temporary region and unless our language is extended with some notion of stack, none of those continuations would be collected until the end of the whole garbage collection. The size of this temporary region can be bounded by the size of the *to* region since we can't allocate more than one continuation per copied object, so it

is still algorithmically efficient, although this memory overhead is a considerable shortcoming.

4.5 Summary and related work

In this chapter, we focussed on a simple stop-and-copy collector. In separate work [MSS01] we have shown how to augment the system presented here to support forwarding pointers and generational collection. In both these cases, type analysis (specifically the Typerec operator on quantified types) is crucial to capturing the invariants that the mutator must satisfy. There are still some more issues that remain to be solved before we can implement a type-safe industrial-strength garbage collector. For example, our generational scheme is feasible only in a language where side-effects are rare. Our scheme also does not handle cyclic data structures.Nevertheless we believe that our current contributions constitute a significant step towards the goal of providing a practical type-safe garbage collector.

Wang and Appel [WA99] proposed to build a tracing garbage collector on top of a region-based calculus, thus providing both type safety and completely automatic memory management. They rely on a closure conversion algorithm due to Tolmach [TO98] that represents closures as datatypes. This makes closures transparent, making it easier for the copy function to analyze, but it requires whole program analysis. We believe it is more natural to represent closures as existentials [MMH96, HM98] and we show how to use intensional analysis of quantified types to typecheck the GC-copy function.

Tofte and Talpin [TT94] proposed to use region calculus to type check memory management for higher-order functional languages. Crary et al [CWM99] presented a low-level typed intermediate language that can express explicit region allocation and deallocation. Our λ_{GC} language borrows the basic organization of memories and regions from Crary et al [CWM99]. The main difference is that we don't require explicit capabilities—region deallocation is handled through the only primitive.

Chapter 5

Integrating Runtime Type Analysis With a Proof System

5.1 Introduction

Until now we have considered type analysis in a system where we can only certify conventional type-safety. However, certifying compilation as originally envisaged by Necula and Lee [NL96, Nec97] through their proof carrying code (PCC) framework, can be used to certify complex specifications [Nec98, AF00a]. For example, the Foundational PCC system [AF00b] can certify any property expressible in Church's higher-order logic.

In this chapter we describe a type system that supports both runtime type analysis and the explicit representation of proofs and propositions. As far as we know, our work is the first comprehensive study on how to integrate higher-order predicate logic and type analysis into typed intermediate languages. Existing type-based certifying compilers [NL98, CLN$^+$00] have focused on simple memory and control-flow safety only. Typed intermediate languages [HM95] and typed assembly languages [MWCG98] also do not rival the expressiveness of the logic used in some PCC systems [AF00b].

This chapter builds upon a large body of previous work in the logic and theorem-proving community [Bar99, Bar91], but makes the following new contributions:

- We show how to design new typed intermediate languages that are capable of representing and manipulating propositions and proofs. We show how these propositions can enforce

more sophisticated program invariants. For example, we can assign an accurate type to unchecked vector (or array) access (see Section 5.5.2). Xi and Pfenning [XP99] can achieve the same using constraint checking, but their system does not support arbitrary propositions and (explicit) proofs, so it is less general than ours.

- We show how to support fully reflexive type analysis in such a type system. We achieve this by using inductive definitions to define the base kind (the kind containing the types of terms). In a sense it generalizes the work presented in Chapter 3.

- We give rigorous proofs for the meta-theoretic properties (subject reduction, strong normalization, confluence, and consistency of the underlying logic) of our type system.

5.2 Approach

Before getting into the details, we first establish a few naming conventions (Figure 5.1). Until now, our typed intermediate languages had three levels. We will now require a fourth level which we call kind schema (*kscm*). We divide the typed intermediate language into a type sub-language and a computation sub-language. The type language contains the top three levels. Kind schemas classify kind terms while kinds classify type terms. We often say that a kind term κ has kind schema u, or a type term τ has kind κ. We assume all kinds used to classify type terms have kind schema Kind, and all types used to classify expressions have kind Ω. For example, both the function type $\tau_1 \rightarrow \tau_2$ and the polymorphic type $\forall\alpha:\kappa.\,\tau$ have kind Ω. Following the tradition, we sometimes say "a kind κ" to imply that κ has kind schema Kind, "a type τ" to imply that τ has kind Ω, and "a type constructor τ" to imply that τ has kind "$\kappa \rightarrow \cdots \rightarrow \Omega$." Kind terms with other kind schemas, or type terms with other kinds are strictly referred to as "kind terms" or "type terms."

The computation language is the lowest level which is where we write the actual program. This language will eventually be compiled into machine code. We often use names such as computation terms, computation values, and computation functions to refer to various constructs at this level.

5.2.1 Representing propositions and proofs

The first step is to represent propositions and proofs for a particular logic in a type-theoretic setting. The most established technique is to use the *formulae-as-types* principle (a.k.a. the Curry-Howard correspondence) [How80] to map propositions and proofs into a typed λ-calculus. The essential

THE TYPE LANGUAGE:

(*kscm*) $u ::= \mathsf{Kind} \mid \ldots$

(*kind*) $\kappa ::= \kappa_1 \rightarrow \kappa_2 \mid \Omega \mid \ldots$

(*type*) $\tau ::= \alpha \mid \lambda\alpha{:}\kappa.\,\tau \mid \tau_1\,\tau_2 \mid \tau_1 \rightarrow \tau_2 \mid \forall\alpha{:}\kappa.\,\tau \mid \ldots$

THE COMPUTATION LANGUAGE:

(*exp*) $e ::= x \mid \lambda x{:}\tau.\,e \mid e_1\,e_2 \mid \Lambda\alpha{:}\kappa.\,e \mid e[\tau] \mid \ldots$

Figure 5.1: Typed intermediate language – notations

idea, which is inspired by constructive logic, is to use types (of kind Ω) to represent propositions, and expressions to represent proofs. A proof of an implication $P \supset Q$ is a function object that yields a proof of proposition Q when applied to a proof of proposition P. A proof of a conjunction $P \wedge Q$ is a pair (e_1, e_2) such that e_1 is a proof of P and e_2 is a proof of Q. A proof of disjunction $P \vee Q$ is a pair (b, e)—a tagged union—where b is either 0 or 1 and if $b = 0$, then e is a proof of P; if $b = 1$ then e is a proof of Q. There is no proof for the false proposition. A proof of a universally quantified proposition $\forall x{\in}B.P(x)$ is a function that maps every element b of the domain B into a proof of $P(b)$ where P is a unary predicate on elements of B. Finally, a proof of an existentially quantified proposition $\exists x{\in}B.P(x)$ is a pair (b, e) where b is an element of B and e is a proof of $P(b)$. Proof-checking in the logic now becomes typechecking in the corresponding typed λ-calculus. There has been a large body of work done along this line in the last 30 years; most type-based proof assistants are based on this fundamental principle. Barendregt *et al.* [Bar99, Bar91] give a good survey on previous work in this area.

Unfortunately, the above scheme fails to work in the context of typed intermediate languages. The problem arises because representing predicates introduces dependent types. For example, suppose Nat is the domain for natural numbers and $Prime$ is a unary predicate that asserts an element of Nat as a prime number. To represent this in a typed setting, we need a type nat representing Nat, and a type constructor prime representing $Prime$. This type constructor must take a number as an argument (to check whether it is prime): therefore the type constructor is dependent on values and has the kind $\mathsf{nat} \rightarrow \Omega$.

Dependent types introduce problems when used in a typed intermediate language. First, real programs often involve effects such as assignment, I/O, or non-termination. Effects interact badly

with dependent types. It is possible to use the effect discipline [SG90] to force types to be dependent on pure computation only, but this does not work in some typed λ-calculi; for example, a "pure" term in Girard's λU [Gir72] could still diverge. Second, many type preserving compilers perform typed CPS conversion [MWCG98], but in the presence of dependent types, this is a very difficult problem [BHS99]. Third, it is important to maintain a phase distinction between compile-time typechecking and run-time evaluation. Having dependent types makes it harder to preserve this property.

5.2.2 Separating the type and computation languages

We solve these problems by making sure that our type language is never dependent on the computation language. Because the actual computation term has to be compiled down to assembly code in any case, it is a bad idea to treat it as part of types. This separation immediately gives us back the phase-distinction property.

To represent propositions and proofs, we lift everything one level up: we use kinds to represent propositions, and type terms for proofs. The domain Nat is represented by a kind Nat; the predicate $Prime$ is represented by a dependent kind term Prime which maps a type term of kind Nat into a proposition. A proof for proposition $\mathsf{Prime}(n)$ certifies that the type term n is a prime number.

To maintain decidable typechecking, we insist that the type language is strongly normalizing and free of side effects. This is possible because the type language no longer depends on any runtime computation. Essentially, we circumvent the problems with dependent types by replacing them with dependent kinds.

To reason about actual programs, we still have to connect terms in the computation language with those in the type language. We follow Xi and Pfenning [XP99] and use singleton types [Hay91] to relate computation values to type terms. In the previous example, we introduce a singleton type constructor snat of kind $\mathsf{Nat} \to \Omega$. Given a type term n of kind Nat, if a computation value v has type $\mathsf{snat}(n)$, then v denotes the natural number represented by n.

A certified prime number package now contains three parts: a type term n of kind Nat, a proof for the proposition $\mathsf{Prime}(n)$, and a computation value of type $\mathsf{snat}(n)$. We can pack it up into an existential package and make it a first-class value with type:

$$\exists n : \mathsf{Nat}.\exists \alpha : \mathsf{Prime}(n).\mathsf{snat}(n).$$

Here we use $\exists$ rather than Σ to emphasize that types and kinds are no longer dependent on computation terms. Under the erasure semantics this certified package is just an integer value of type $\mathsf{snat}(n)$ at run time.

We can also certify programs that involve effects. Assume again that f is a function in the computation language which may not terminate on some inputs. Suppose we want to certify that if the input to f is a prime, and the call to f does return, then the result is also a prime. We can achieve this in two steps. First, we construct a type-level function g of kind $\mathsf{Nat} \rightarrow \mathsf{Nat}$ to simulate the behavior of f (on all inputs where f does terminate) and show that f has the following type:

$$\forall n{:}\mathsf{Nat}.\ \mathsf{snat}(n) \rightarrow \mathsf{snat}(g(n))$$

Here following Figure 5.1, we use $\forall$ and $\rightarrow$ to denote the polymorphic and function types for the computation language. The type for f says that if it takes an integer of type $\mathsf{snat}(n)$ as input and does not loop forever, then it will return an integer of type $\mathsf{snat}(g(n))$. Second, we construct a proof τ_p showing that g always maps a prime to another prime. The certified binary for f now also contains three parts: the type-level function g, the proof τ_p, and the computation function f itself. We can pack it into an existential package with type:

$$\exists g{:}\mathsf{Nat} \rightarrow \mathsf{Nat}.\ \exists p{:}(\Pi t{:}\mathsf{Nat}.\mathsf{Prime}(t) \rightarrow \mathsf{Prime}(g(t))).$$
$$\forall n{:}\mathsf{Nat}.\ \mathsf{snat}(n) \rightarrow \mathsf{snat}(g(n))$$

Notice this type also contains function applications such as $g(n)$, but g is a type-level function which is always strongly normalizing, so typechecking is still decidable.

5.2.3 Designing the type language

We can incorporate propositions and proofs into a type system, but in addition the type language must fulfill its usual responsibilities. First, it must provide a set of types (of kind Ω) to classify the computation terms. Second, it must support the intensional analysis of these types.

Our approach to this is to generalize the solution (for type analysis) given in Chapter 3. There we introduced kind polymorphism so that the types could be defined in a way that made the base kind inductive. Here we will go a step further and provide a general mechanism of defining inductive kinds. The base kind Ω is then defined inductively using this mechanism. Inductive definitions also greatly increase the programming power of our type language. We can introduce new data ob-

$$
\begin{array}{ll}
(\textit{kscm}) & u ::= z \mid \Pi\alpha\!:\!\kappa.\, u \mid \Pi j\!:\!u_1.\, u_2 \mid \mathsf{Kind} \\
(\textit{kind}) & \kappa ::= j \mid \lambda\alpha\!:\!\kappa_1.\, \kappa_2 \mid \kappa[\tau] \mid \lambda j\!:\!u.\, \kappa \mid \kappa_1\, \kappa_2 \mid \Pi\alpha\!:\!\kappa_1.\, \kappa_2 \mid \Pi j\!:\!u.\, \kappa \\
& \quad \mid \Pi z\!:\!\mathsf{Kscm}.\, \kappa \mid \mathsf{Ind}(j\!:\!\mathsf{Kind})\{\vec{\kappa}\} \mid \mathsf{Elim}[\kappa', u](\tau)\{\vec{\kappa}\} \\
(\textit{type}) & \tau ::= \alpha \mid \lambda\alpha\!:\!\kappa.\, \tau \mid \tau_1\, \tau_2 \mid \lambda j\!:\!u.\, \tau \mid \tau[\kappa] \mid \lambda z\!:\!\mathsf{Kscm}.\, \tau \mid \tau[u] \\
& \quad \mid \mathsf{Ctor}\,(i, \kappa) \mid \mathsf{Elim}[\kappa', \kappa](\tau')\{\vec{\tau}\}
\end{array}
$$

Figure 5.2: Syntax of λ^i_{CC}

jects (*e.g.*, integers, lists) and define primitive recursive functions, all at the type level; these in turn are used to help model the behaviors of the computation terms.

In the rest of this chapter, we first give a formal definition of our type language (which will be named as λ^i_{CC} from now on) in Section 5.3. To show how this type system can be used, we then present a sample computation language λ_H in Section 5.5.

5.3 The type language λ^i_{CC}

Our type language λ^i_{CC} resembles the calculus of inductive constructions (CIC) implemented in the Coq proof assistant [HPM+00]. We do not directly use CIC as our type language for the following reasons: first, CIC contains some features designed for program extraction [Pau89] which are not required in our case (where proofs are only used as specifications for the computation terms). Second, as far as we know, there are still no formal studies covering the entire CIC language.

The syntax for λ^i_{CC} is shown in Figure 5.2. Here kind schemas (*kscm*) classify kind terms while kinds classify type terms. There are variables at all three levels: kind-schema variables z, kind variables j, and type variables α. We have an external constant Kscm classifying all the kind schemas; essentially, λ^i_{CC} has an additional level above *kscm*, of which Kscm is the sole member.

A good way to comprehend λ^i_{CC} is to look at its five Π constructs: there are three at the kind level and two at the kind-schema level. Each Π term is used to typecheck a λ-function and its application form defined at a level below. We use a few examples to explain why each of them is necessary. Following the tradition, we use arrow terms (*e.g.*, $\kappa_1 \to \kappa_2$) as a syntactic sugar for the non-dependent Π terms (*e.g.*, $\Pi\alpha\!:\!\kappa_1.\, \kappa_2$ is non-dependent if α does not occur free in κ_2).

- Kinds $\Pi\alpha : \kappa_1.\, \kappa_2$ and $\kappa_1 \to \kappa_2$ are used to typecheck the type-level function $\lambda\alpha : \kappa.\, \tau$ and its application form $\tau_1\, \tau_2$. Assuming Ω and Nat are inductive kinds (defined later), we can write a type term such as $\lambda\alpha\!:\!\Omega.\, \alpha$ which has kind $\Omega \to \Omega$, or a type-level arithmetic function

such as plus which has kind $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$.

- Kinds $\Pi j : u.\, \kappa$ and $u \to \kappa$ are used to typecheck the type-level kind abstraction $\lambda j : u.\, \tau$ and its application form $\tau[\kappa]$. As we saw before, this is needed to support intensional analysis of quantified types. It can also be used to define logic connectives and constants, *e.g.*

$$\begin{aligned} \mathsf{True} \ : \ \mathsf{Kind} &= \Pi j : \mathsf{Kind}.\, j \to j \\ \mathsf{False} \ : \ \mathsf{Kind} &= \Pi j : \mathsf{Kind}.\, j \end{aligned}$$

  True has the polymorphic identity as a proof:

$$\mathsf{id} \ : \ \mathsf{True} \quad = \quad \lambda j : \mathsf{Kind}.\, \lambda \alpha : j.\, \alpha$$

  but False is not inhabited (this is essentially the consistency property of λ^{i}_{CC} which we will show later).

- Kind $\Pi z : \mathsf{Kscm}.\, \kappa$ is used to typecheck the type-level kind-schema abstraction $\lambda z : \mathsf{Kscm}.\, \tau$ and its application form $\tau[u]$. This is not in the core calculus of constructions [CH88]. We use it in the inductive definition of Ω (see Section 5.5) where both the $\forall_{\mathsf{Kscm}}$ and $\exists_{\mathsf{Kscm}}$ constructors have kind $\Pi z : \mathsf{Kscm}.\, (z \to \Omega) \to \Omega$. These two constructors in turn allow us to typecheck predicate-polymorphic computation terms, which occurs during closure conversion [SSTP01].

- Kind schemas $\Pi \alpha : \kappa.\, u$ and $\kappa \to u$ are used to typecheck the kind-level type abstraction $\lambda \alpha : \kappa_1.\, \kappa_2$ and its application form $\kappa[\tau]$. The predicate Prime has kind schema $\mathsf{Nat} \to \mathsf{Kind}$. A predicate with kind schema $\Pi \alpha : \mathsf{Nat}.\, \mathsf{Prime}(\alpha) \to \mathsf{Kind}$ is only applicable to prime numbers. We can also define *e.g.* a binary relation:

$$\mathsf{LT} \ : \ \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Kind}$$

  so that LT α_1 α_2 is a proposition asserting that the natural number represented by α_1 is less than that of α_2.

- Kind schemas $\Pi j : u_1.\, u_2$ and $u_1 \to u_2$ are used to typecheck the kind-level function $\lambda j : u.\, \kappa$ and its application form $\kappa_1\, \kappa_2$. We use it to write higher-order predicates and logic

connectives. For example, the logical negation operator can be written as follows:

$$\mathsf{Not} \ : \ \mathsf{Kind} \to \mathsf{Kind} \quad = \quad \lambda j\!:\!\mathsf{Kind}.\,(j \to \mathsf{False})$$

The consistency of λ^i_{CC} implies that a proposition and its negation cannot be both inhabited—otherwise applying the proof of the second to that of the first would yield a proof of False.

λ^i_{CC} also provides a general mechanism of inductive definitions [Pau93]. The term $\mathsf{Ind}(j : \mathsf{Kind})\{\vec{\kappa}\}$ introduces an inductively defined kind j containing a list of constructors whose kinds are specified by $\vec{\kappa}$. Here j must only occur "positively" (Section 5.4) inside each κ_i. The term $\mathsf{Ctor}\,(i, \kappa)$ refers to the i-th constructor in an inductive kind κ. For presentation, we will use a more friendly syntax in the rest of this chapter. An inductive kind $I = \mathsf{Ind}(j\!:\!\mathsf{Kind})\{\vec{\kappa}\}$ will be written as shown below. We give an explicit name c_i to each constructor, so c_i is just an abbreviation of $\mathsf{Ctor}\,(i, I)$.

$$\begin{aligned} \mathsf{Inductive}\ I \ : \ \mathsf{Kind} \ := \mathsf{c}_1 \ &: [I/j]\kappa_1 \\ \mid \mathsf{c}_2 \ &: [I/j]\kappa_2 \\ &\vdots \\ \mid \mathsf{c}_n \ &: [I/j]\kappa_n \end{aligned}$$

λ^i_{CC} provides two iterators to support primitive recursion on inductive kinds. The small elimination $\mathsf{Elim}[\kappa', \kappa](\tau')\{\vec{\tau}\}$ takes a type term τ' of inductive kind κ', performs the iterative operation specified by $\vec{\tau}$ (which contains a branch for each constructor of κ'), and returns a type term of kind $\kappa[\tau']$ as the result. The large elimination $\mathsf{Elim}[\kappa', u](\tau)\{\vec{\kappa}\}$ takes a type term τ of inductive kind κ', performs the iterative operation specified by $\vec{\kappa}$, and returns a kind term of kind schema u as the result. These iterators generalize the $\mathsf{Typerec}$ operator defined in Chapter 3.

Figure 5.3 gives a few examples of inductive definitions including the inductive kinds Bool and Nat and several type-level functions which we will use in Section 5.5. The small elimination for Nat takes the following form $\mathsf{Elim}[\mathsf{Nat}, \kappa](\tau')\{\tau_1; \tau_2\}$. Here, κ is a dependent kind with kind schema $\mathsf{Nat} \to \mathsf{Kind}$; τ' is the argument which has kind Nat. The term in the zero branch, τ_1, has kind $\kappa[\tau']$. The term in the succ branch, τ_2, has kind $\mathsf{Nat} \to \kappa[\tau'] \to \kappa[\tau']$. We denote the iterator operation in λ^i_{CC} as the ι-reduction. For example, the two ι-reduction rules for Nat work

$$\begin{array}{l}
\mathsf{Inductive\ Bool} : \mathsf{Kind} := \mathsf{true} : \mathsf{Bool} \\
\qquad\qquad\qquad\qquad\quad \mid \mathsf{false} : \mathsf{Bool} \\
\mathsf{Inductive\ Nat} : \mathsf{Kind} := \mathsf{zero} : \mathsf{Nat} \\
\qquad\qquad\qquad\qquad\quad \mid \mathsf{succ} : \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{plus} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{plus}(\mathsf{zero}) \quad = \quad \lambda\alpha{:}\mathsf{Nat}.\,\alpha \\
\mathsf{plus}(\mathsf{succ}\ \alpha) \quad = \quad \lambda\alpha'{:}\mathsf{Nat}.\,\mathsf{succ}\ ((\mathsf{plus}\ \alpha)\ \alpha') \\
\mathsf{ifez} : \mathsf{Nat} \to (\Pi j{:}\mathsf{Kind}.\, j \to (\mathsf{Nat} \to j) \to j) \\
\mathsf{ifez}(\mathsf{zero}) \quad = \quad \lambda j{:}\mathsf{Kind}.\,\lambda\alpha_1{:}j.\,\lambda\alpha_2{:}\mathsf{Nat} \to j.\,\alpha_1 \\
\mathsf{ifez}(\mathsf{succ}\ \alpha) \quad = \quad \lambda j{:}\mathsf{Kind}.\,\lambda\alpha_1{:}j.\,\lambda\alpha_2{:}\mathsf{Nat} \to j.\,\alpha_2\ \alpha \\
\mathsf{le} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Bool} \\
\mathsf{le}(\mathsf{zero}) \quad = \quad \lambda\alpha{:}\mathsf{Nat}.\,\mathsf{true} \\
\mathsf{le}(\mathsf{succ}\ \alpha) \quad = \quad \lambda\alpha'{:}\mathsf{Nat}.\,\mathsf{ifez}\ \alpha'\ \mathsf{Bool}\ \mathsf{false}\ (\mathsf{le}\ \alpha) \\
\mathsf{lt} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Bool} \\
\mathsf{lt} \quad = \quad \lambda\alpha{:}\mathsf{Nat}.\,\mathsf{le}\ (\mathsf{succ}\ \alpha) \\
\mathsf{Cond} : \mathsf{Bool} \to \mathsf{Kind} \to \mathsf{Kind} \to \mathsf{Kind} \\
\mathsf{Cond}(\mathsf{true}) \quad = \quad \lambda j_1{:}\mathsf{Kind}.\,\lambda j_2{:}\mathsf{Kind}.\,j_1 \\
\mathsf{Cond}(\mathsf{false}) \quad = \quad \lambda j_1{:}\mathsf{Kind}.\,\lambda j_2{:}\mathsf{Kind}.\,j_2
\end{array}$$

Figure 5.3: Examples of inductive definitions

as follows:

$$\mathsf{Elim}[\mathsf{Nat}, \kappa](\mathsf{zero})\{\tau_1; \tau_2\} \leadsto_\iota \tau_1$$

$$\mathsf{Elim}[\mathsf{Nat}, \kappa](\mathsf{succ}\ \tau)\{\tau_1; \tau_2\} \leadsto_\iota \tau_2\ \tau\ (\mathsf{Elim}[\mathsf{Nat}, \kappa](\tau)\{\tau_1; \tau_2\})$$

In Figure 5.3, plus is a function which calculates the sum of two natural numbers. The function ifez behaves like a switch statement: if its argument is zero, it returns a function that selects the first branch; otherwise, the result takes the second branch and applies it to the predecessor of the argument. The function le evaluates to true if its first argument is less than or equal to the second. The function lt performs the less-than comparison.

The definition of function Cond, which implements a conditional with result at the kind level, uses large elimination on Bool. It has the form $\mathsf{Elim}[\mathsf{Bool}, u](\tau)\{\kappa_1; \kappa_2\}$, where τ is of kind Bool; both the true and false branches (κ_1 and κ_2) have kind schema u.

5.4 Formalization of λ^i_{CC}

In this section, we formalize our type language. It is easier to do this in terms of a PTS specification (Section 2.5). Notice from Figure 5.2 that all the three layers in our type language essentially consist only of abstractions, applications, and constructs related to inductive definitions. A PTS specification allows us to factor out this commonality.

The syntax for the PTS pseudoterms is:

$$
\begin{array}{lll}
(\textit{ctxt}) & \Delta & ::= \cdot \mid \Delta, X : A \\
(\textit{sort}) & s & ::= \mathsf{Kind} \mid \mathsf{Kscm} \mid \mathsf{Ext} \\
(\textit{var}) & X & ::= z \mid j \mid \alpha \\
(\textit{ptm}) & A, B & ::= s \mid X \mid \lambda X : A.\, B \mid A\ B \mid \Pi X : A.\, B \mid \mathsf{Ind}(X : \mathsf{Kind})\{\vec{A}\} \\
 & & \quad \mid \mathsf{Ctor}\,(i, A) \mid \mathsf{Elim}[A', B'](A)\{\vec{B}\}
\end{array}
$$

In addition to the symbols defined in the syntax, we will also use C to denote general terms, Y and Z for variables, and I for inductive definitions. We use $\vec{A}$ to denote a sequence of terms $A_1,\ldots,A_n$. Also, we distinguish between A and $\vec{A}$ since every element in $\vec{A}$ would be referred as A_i anyway.

λ^i_{CC} has the following PTS specification which will be used to derive its typing rules:

$$
\begin{array}{lcl}
\mathcal{S} & = & \mathsf{Kind}, \mathsf{Kscm}, \mathsf{Ext} \\
\mathcal{A} & = & \mathsf{Kind} : \mathsf{Kscm}, \mathsf{Kscm} : \mathsf{Ext} \\
\mathcal{R} & = & (\mathsf{Kind}, \mathsf{Kind}), (\mathsf{Kscm}, \mathsf{Kind}), (\mathsf{Ext}, \mathsf{Kind}) \\
 & & (\mathsf{Kind}, \mathsf{Kscm}), (\mathsf{Kscm}, \mathsf{Kscm})
\end{array}
$$

In order to ensure that the interpretation of inductive definitions remains consistent, and they can be interpreted as terms closed under their introduction rules, we impose *positivity constraints* on the constructors of an inductive definition. The positivity constraints are defined in Definition 5.4.1 and 5.4.2.

Definition 5.4.1 *A term A is* strictly positive *in X if A is either X or $\Pi Y : B.\, A'$, where A' is strictly positive in X, X does not occur free in B, and $X \neq Y$.*

Definition 5.4.2 *A term C is a* well-formed constructor kind *for X (written $wfc_X(C)$) if it has one of the following forms:*

1. X;

2. $\Pi Y : B.\, C'$, *where* $Y \neq X$, X *is not free in* B, *and* C' *is a well-formed constructor kind for* X; *or*

3. $A \rightarrow C'$, *where* A *is strictly positive in* X *and* C' *is a well-formed constructor kind for* X.

Note that in the definition of $wfc_X(C)$, the second clause covers the case where C is of the form $A \rightarrow C'$, and X does not occur free in A. Therefore, we only allow the occurrence of X in the non-dependent case.

We often write the well-formed constructor kind for X as $\Pi \vec{Y} : \vec{B}.\, X$. We also denote terms that are strictly positive in X by $\Pi \vec{Y} : \vec{B}.\, X$, where X is not free in $\vec{B}$.

Definition 5.4.3 *Let* C *be a well-formed constructor kind for* X. *Then* C *is of the form* $\Pi \vec{Y} : \vec{A}.\, X$. *If all the* Y*'s are* α*'s, that is,* C *is of the form* $\Pi \vec{\alpha} : \vec{A}.\, X$, *then we say that* C *is a small constructor kind (or just small constructor when there is no ambiguity) and denote it as* $small(C)$.

Our inductive definitions reside in Kind, whereas a small constructor does not make universal quantification over objects of type Kind. Therefore, an inductive definition with small constructors is a predicative definition. While dealing with impredicative inductive definitions, we must forbid projections on universes equal to or bigger than the one inhabited by the definition. In particular, we restrict large elimination to inductive definitions with only small constructors.

Next, we define the set of reductions on our terms. The definition of β- and η-reduction is standard. The ι-reduction defines primitive recursion over inductive objects.

Definition 5.4.4 *Let* C *be a well-formed constructor kind for* X *and let* A', B', *and* I *be pseudoterms. We define* $\Phi_{X,I,B'}(C, A')$ *recursively based on the structure of* C:

$$
\begin{array}{lcl}
\Phi_{X,I,B'}(X, A') & \stackrel{\text{def}}{=} & A' \\
\Phi_{X,I,B'}(\Pi Y : B.\, C', A') & \stackrel{\text{def}}{=} & \lambda Y : B.\, \Phi_{X,I,B'}(C', A'\ Y) \\
\Phi_{X,I,B'}((\Pi \vec{Y} : \vec{B}.\, X) \rightarrow C', A') & \stackrel{\text{def}}{=} & \lambda Z : (\Pi \vec{Y} : \vec{B}.\, I).\, \Phi_{X,I,B'}(C', A'\ Z\ (\lambda \vec{Y} : \vec{B}.\, B'\ (Z\ \vec{Y})))
\end{array}
$$

Definition 5.4.5 *The reduction relations on our terms are defined as:*

$$\begin{array}{rcl}
(\lambda X\!:\!A.\,B)\ A' & \rightsquigarrow_\beta & [A'/X]B \\
\lambda X\!:\!A.\,(B\ X) & \rightsquigarrow_\eta & B, \quad \textit{if } X \notin FV(B) \\
\mathsf{Elim}[I, A'']\big(\mathsf{Ctor}\,(i, I)\ \vec{A}\big)\{\vec{B}\} & \rightsquigarrow_\iota & (\Phi_{X,I,B'}(C_i, B_i))\ \vec{A} \\
\multicolumn{3}{l}{\textit{where} \quad I = \mathsf{Ind}(X\!:\!\mathsf{Kind})\{\vec{C}\}} \\
\multicolumn{3}{l}{\qquad\quad B' = \lambda Y\!:\!I.\,(\mathsf{Elim}[I, A''](Y)\{\vec{B}\})}
\end{array}$$

By $\rhd_\beta$, $\rhd_\eta$, and $\rhd_\iota$ we denote the relations that correspond to the rewriting of subterms using the relations $\rightsquigarrow_\beta$, $\rightsquigarrow_\eta$, and $\rightsquigarrow_\iota$ respectively. We use $\rightsquigarrow$ and $\rhd$ for the unions of the above relations. We also write $\rhd^*$ and $\rhd^+$ (respectively $\rhd^*_\beta$ etc.) for the reflexive-transitive and transitive closures of $\rhd$ (respectively $\rhd_\beta$ etc.) and $=_{\beta\eta\iota}$ for the reflexive-symmetric-transitive closure of $\rhd$. We say that a sequence of terms $A_1,\ldots,A_n$, such that $A \rhd A_1 \rhd A_2 \ldots \rhd A_n$, is a chain of reductions starting from A.

Let us examine the ι-reduction in detail. In $\mathsf{Elim}[I, A''](A)\{\vec{B}\}$, the term A of type I is being analyzed. The sequence $\vec{B}$ contains the set of branches for Elim, one for each constructor of I. In the case when $C_i = X$, which implies that A is of the form $\mathsf{Ctor}\,(i, I)$, the Elim just selects the B_i branch:

$$\mathsf{Elim}[I, A'']\big(\mathsf{Ctor}\,(i, I)\big)\{\vec{B}\} \quad \rightsquigarrow_\iota \quad B_i$$

In the case when $C_i = \Pi\vec{Y} : \vec{B}.\,X$ where X does not occur free in $\vec{B}$, then A must be in the form $\mathsf{Ctor}\,(i, I)\ \vec{A}$ with A_i of type B_i. None of the arguments are recursive. Therefore, the Elim should just select the B_i branch and pass the constructor arguments to it. Accordingly, the reduction yields (by expanding the Φ macro):

$$\mathsf{Elim}[I, A'']\big(\mathsf{Ctor}\,(i, I)\ \vec{A}\big)\{\vec{B}\} \quad \rightsquigarrow_\iota \quad B_i\ \vec{A}$$

The recursive case is the most interesting. For simplicity assume that the i-th constructor has the form $\Pi\vec{Y} : \vec{B'}.\,X \to \Pi\vec{Y'} : \vec{B''}.\,X$. Therefore, A is of the form $\mathsf{Ctor}\,(i, I)\ \vec{A}$ with A_1 being the recursive component of type $\Pi\vec{Y} : \vec{B'}.\,X$, and $A_2 \ldots A_n$ being non-recursive. The reduction rule then yields:

$$\mathsf{Elim}[I, A'']\big(\mathsf{Ctor}\,(i, I)\ \vec{A}\big)\{\vec{B}\} \rightsquigarrow_\iota B_i\ A_1\ (\lambda\vec{Y}\!:\!\vec{B'}.\,\mathsf{Elim}[I, A''](A_1\ \vec{Y})\{\vec{B}\})\ A_2 \ldots A_n$$

The Elim construct selects the B_i branch and passes the arguments $A_1,\ldots,A_n$, and the result of recursively processing A_1. In the general case, it would process each recursive argument.

Definition 5.4.6 defines the Ψ macro which represents the type of the large Elim branches. Definition 5.4.7 defines the ζ macro which represents the type of the small elimination branches. The different cases follow from the ι-reduction rule in Definition 5.4.5.

Definition 5.4.6 *Let C be a well-formed constructor kind for X and let A' and I be two terms. We define $\Psi_{X,I}(C, A')$ recursively based on the structure of C:*

$$\begin{aligned}
\Psi_{X,I}(X, A') &\stackrel{\text{def}}{=} A' \\
\Psi_{X,I}(\Pi Y : B.\, C', A') &\stackrel{\text{def}}{=} \Pi Y : B.\, \Psi_{X,I}(C', A') \\
\Psi_{X,I}(A \to C', A') &\stackrel{\text{def}}{=} [I/X]A \to [A'/X]A \to \Psi_{X,I}(C', A')
\end{aligned}$$

where X is not free in B and A is strictly positive in X.

Definition 5.4.7 *Let C be a well-formed constructor kind for X and let A', I, and B' be terms. We define $\zeta_{X,I}(C, A', B')$ recursively based on the structure of C:*

$$\begin{aligned}
\zeta_{X,I}(X, A', B') &\stackrel{\text{def}}{=} A'\ B' \\
\zeta_{X,I}(\Pi Y : B.\, C', A', B') &\stackrel{\text{def}}{=} \Pi Y : B.\, \zeta_{X,I}(C', A', B'\ Y) \\
\zeta_{X,I}(\Pi \vec{Y} : \vec{B}.\, X \to C', A', B') &\stackrel{\text{def}}{=} \Pi Z : (\Pi \vec{Y} : \vec{B}.\, I).\, \Pi \vec{Y} : \vec{B}.\, (A'\ (Z\ \vec{Y})) \to \zeta_{X,I}(C', A', B'\ Z)
\end{aligned}$$

where X is not free in B and $\vec{B}$.

Definition 5.4.8 *We use $\Delta|_{\alpha,j}$ to denote that the environment does not contain any z variables.*

Here are the complete typing rules for λ^{i}_{CC}. The three weakening rules make sure that all variables are bound to the right classes of terms in the context. There are no separate context-formation rules; a context Δ is well-formed if we can derive the judgment $\Delta \vdash \mathsf{Kind} : \mathsf{Kscm}$ (notice we can only add new variables to the context via the weakening rules).

$$\cdot \vdash \mathsf{Kind} : \mathsf{Kscm} \qquad (\text{AX}1)$$

$$\cdot \vdash \mathsf{Kscm} : \mathsf{Ext} \qquad (\text{AX}2)$$

$$\frac{\Delta \vdash C : \mathsf{Kind} \quad \Delta \vdash A : B \quad \alpha \notin Dom(\Delta)}{\Delta, \alpha : C \vdash A : B} \qquad (\text{WEAK}1)$$

$$\frac{\Delta \vdash C : \mathsf{Kscm} \quad \Delta \vdash A : B \quad j \notin Dom(\Delta)}{\Delta, j{:}C \vdash A : B} \quad (\textsc{weak}2)$$

$$\frac{\Delta \vdash C : \mathsf{Ext} \quad \Delta \vdash A : B \quad z \notin Dom(\Delta)}{\Delta, z{:}C \vdash A : B} \quad (\textsc{weak}3)$$

$$\frac{\Delta \vdash \mathsf{Kind} : \mathsf{Kscm} \qquad X \in Dom(\Delta)}{\Delta \vdash X : \Delta(X)} \quad (\textsc{var})$$

$$\frac{\Delta, X{:}A \vdash B : B' \qquad \Delta \vdash \Pi X{:}A.\, B' : s}{\Delta \vdash \lambda X{:}A.\, B : \Pi X{:}A.\, B'} \quad (\textsc{fun})$$

$$\frac{\Delta \vdash A : \Pi X{:}B'.\, A' \quad \Delta \vdash B : B'}{\Delta \vdash A\ B : [B/X]A'} \quad (\textsc{app})$$

$$\frac{\Delta \vdash A : s_1 \quad \Delta, X{:}A \vdash B : s_2 \quad (s_1, s_2) \in \mathcal{R}}{\Delta \vdash \Pi X{:}A.\, B : s_2} \quad (\textsc{prod})$$

$$\frac{\textit{for all } i \qquad \Delta, X{:}\mathsf{Kind} \vdash C_i : \mathsf{Kind} \qquad wfc_X(C_i)}{\Delta \vdash \mathsf{Ind}(X{:}\mathsf{Kind})\{\vec{C}\} : \mathsf{Kind}} \quad (\textsc{ind})$$

$$\frac{\Delta \vdash I : \mathsf{Kind} \textit{ where } I = \mathsf{Ind}(X{:}\mathsf{Kind})\{\vec{C}\}}{\Delta \vdash \mathsf{Ctor}\,(i, I) : [I/X]C_i} \quad (\textsc{con})$$

$$\frac{\begin{array}{c}\Delta \vdash A : I \qquad \Delta \vdash A' : I \to \mathsf{Kind} \\ \textit{for all } i \qquad \Delta \vdash B_i : \zeta_{X,I}(C_i, A', \mathsf{Ctor}\,(i, I))\end{array}}{\begin{array}{c}\Delta \vdash \mathsf{Elim}[I, A'](A)\{\vec{B}\} : A'\ A \\ \textit{where } I = \mathsf{Ind}(X{:}\mathsf{Kind})\{\vec{C}\}\end{array}} \quad (\textsc{elim})$$

$$\frac{\begin{array}{c}\Delta \vdash A : I \qquad \Delta|_{\alpha,j} \vdash A' : \mathsf{Kscm} \\ \textit{for all } i \quad small(C_i) \quad \Delta \vdash B_i : \Psi_{X,I}(C_i, A')\end{array}}{\begin{array}{c}\Delta \vdash \mathsf{Elim}[I, A'](A)\{\vec{B}\} : A' \\ \textit{where } I = \mathsf{Ind}(X{:}\mathsf{Kind})\{\vec{C}\}\end{array}} \quad (\textsc{l-elim})$$

$$\frac{\Delta \vdash A : B \qquad \Delta \vdash B' : s \qquad \Delta \vdash B : s \qquad B =_{\beta\eta\iota} B'}{\Delta \vdash A : B'} \quad (\textsc{conv})$$

We will merely state λ^{i}_{CC}'s meta-theoretic properties here and defer the proofs to Appendix C.

Theorem 5.4.9 *Reductions of well-formed terms is strongly normalizing and confluent.*

We will next show a sample computation language and show how the expressivity of the type language may be put to use.

$$
\begin{array}{llll}
(\textit{exp}) & e & ::= & x \mid \overline{n} \mid \mathsf{tt} \mid \mathsf{ff} \mid f \mid \mathsf{fix}\, x\!:\!A.\, f \mid e\ e' \mid e[A] \mid \langle X = A,\ e\!:\!A' \rangle \\
& & & \quad \mid \mathsf{open}\ e\ \mathsf{as}\ \langle X,\ x \rangle\ \mathsf{in}\ e' \mid \langle e_0,\ \ldots\ e_{n-1} \rangle \mid \mathsf{sel}[A](e, e') \mid e\ \mathit{aop}\ e' \\
& & & \quad \mid e\ \mathit{cop}\ e' \mid \mathsf{if}[A, A']\mathsf{(}e,\ X_1.\, e_1,\ X_2.\, e_2\mathsf{)} \\
& & & \text{where } n \in \mathbf{N} \\
(\textit{fun}) & f & ::= & \lambda x\!:\!A.\, e \mid \Lambda X\!:\!A.\, f \\
(\textit{arith}) & \mathit{aop} & ::= & \mathsf{+} \mid \ldots \\
(\textit{cmp}) & \mathit{cop} & ::= & \mathsf{<} \mid \ldots
\end{array}
$$

Figure 5.4: Syntax of the computation language λ_H.

5.5 The computation language λ_H

The language of computations λ_H can use proofs and propositions (constructed in the type language) to represent program invariants expressible in higher-order predicate logic. This allows us to assign a more refined type to programs when compared to other higher-order typed calculi.

In this section we often use the unqualified "term" to refer to a computation term (expression) e, with syntax defined in Figure 5.4. Most of the constructs are borrowed from standard higher-order typed calculi. We will only consider constants representing natural numbers ($\overline{n}$ is the value representing $n \in \mathbf{N}$) and boolean values (tt and ff). The term-level abstraction and application are standard; type abstractions and fixed points are restricted to function values since we use call-by-value semantics. The type variable bound by a type abstraction, as well as the one bound by the open construct for packages of existential type, can have either a kind or a kind schema. Dually, the type argument in a type application, and the witness type term A in the package construction $\langle X = A,\ e\!:\!A' \rangle$ can be either a type term or a kind term.

The constructs implementing tuple operations, arithmetic, and comparisons have nonstandard static semantics, on which we focus in section 5.5.1, but their runtime behavior is standard. The branching construct is parameterized at the type level with a proposition (which is dependent on the value of the test term) and its proof; the proof is passed to the executed branch.

Dynamic semantics We present a small step call-by-value operational semantics for λ_H in the style of Wright and Felleisen [WC94]. The values are defined as shown below. The reduction

$$(\lambda x\!:\!A.\, e)\ v \ \leadsto\ [v/x]e \qquad \text{(R-}\beta\text{)}$$

$$(\Lambda X\!:\!B.\, f)[A] \ \leadsto\ [A/X]f \qquad \text{(R-TY-}\beta\text{)}$$

$$\mathsf{sel}[A](\langle v_0,\, \ldots\ v_{n-1}\rangle, \overline{m}) \ \leadsto\ v_m \qquad (m < n) \qquad \text{(R-SEL)}$$

$$\mathsf{open}\ \langle X' = A,\, v\!:\!A'\rangle\ \mathsf{as}\ \langle X,\, x\rangle\ \mathsf{in}\ e \ \leadsto\ [v/x][A/X]e \qquad \text{(R-OPEN)}$$

$$(\mathsf{fix}\ x\!:\!A.\, f)\ v \ \leadsto\ ([\mathsf{fix}\ x\!:\!A.\, f/x]f)\ v \qquad \text{(R-FIX)}$$

$$(\mathsf{fix}\ x\!:\!A.\, f)[A'] \ \leadsto\ ([\mathsf{fix}\ x\!:\!A.\, f/x]f)[A'] \qquad \text{(R-TYFIX)}$$

$$\overline{m} + \overline{n} \ \leadsto\ \overline{m+n} \qquad \text{(R-ADD)}$$

$$\overline{m} < \overline{n} \ \leadsto\ \mathsf{tt} \qquad (m < n) \qquad \text{(R-LT-T)}$$

$$\overline{m} < \overline{n} \ \leadsto\ \mathsf{ff} \qquad (m \geq n) \qquad \text{(R-LT-F)}$$

$$\mathsf{if}[B, A](\mathsf{tt},\, X_1.\, e_1,\, X_2.\, e_2) \ \leadsto\ [A/X_1]e_1 \qquad \text{(R-IF-T)}$$

$$\mathsf{if}[B, A](\mathsf{ff},\, X_1.\, e_1,\, X_2.\, e_2) \ \leadsto\ [A/X_2]e_2 \qquad \text{(R-IF-F)}$$

Figure 5.5: Dynamic semantics

relation $\leadsto$ is specified in Figure 5.5.

$$v ::= \overline{n} \mid \mathsf{tt} \mid \mathsf{ff} \mid f \mid \mathsf{fix}\ x\!:\!A.\, f \mid \langle X = A,\, v\!:\!A'\rangle \mid \langle v_0,\, \ldots\ v_{n-1}\rangle$$

An evaluation context E encodes the call-by-value discipline:

$$\begin{aligned} E ::= {} & \bullet \mid E\ e \mid v\ E \mid E[A] \mid \langle X = A,\, E\!:\!A'\rangle \mid \mathsf{open}\ E\ \mathsf{as}\ \langle X,\, x\rangle\ \mathsf{in}\ e \\ & \mid \mathsf{open}\ v\ \mathsf{as}\ \langle X,\, x\rangle\ \mathsf{in}\ E \mid \langle v_0, \ldots v_i,\, E,\, e_{i+2},\, \ldots,\, e_{n-1}\rangle \mid \mathsf{sel}[A](E, e) \\ & \mid \mathsf{sel}[A](v, E) \mid E\ aop\ e \mid v\ aop\ E \mid E\ cop\ e \mid v\ cop\ E \\ & \mid \mathsf{if}[A, A']\mathsf{(}E,\, X_1.\, e_1,\, X_2.\, e_2\mathsf{)} \end{aligned}$$

The notation $E\{e\}$ stands for the term obtained by replacing the hole $\bullet$ in E by e. The single step computation $\mapsto$ relates $E\{e\}$ to $E\{e'\}$ when $e \leadsto e'$, and $\mapsto^*$ is its reflexive transitive closure.

As shown the semantics is standard except for some additional passing of type terms in R-SEL and R-IF-T/F. However an inspection of the rules shows that types are irrelevant for the evaluation, hence a type-erasure semantics, in which all type-related operations and parameters are erased, would be entirely standard.

5.5.1 Static semantics

The static semantics of λ_H shows the benefits of using a type language as expressive as λ^i_{CC}. We can now define the type constructors of λ_H as constructors of an inductive kind Ω, instead of having them built into λ_H.

$$\begin{aligned}
\mathsf{Inductive}\ \Omega : \mathsf{Kind} := \mathsf{snat} &: \mathsf{Nat} \to \Omega \\
\mid \mathsf{sbool} &: \mathsf{Bool} \to \Omega \\
\mid \twoheadrightarrow &: \Omega \to \Omega \to \Omega \\
\mid \ll &: \mathsf{Nat} \to (\mathsf{Nat} \to \Omega) \to \Omega \\
\mid \forall_{\mathsf{Kind}} &: \Pi j : \mathsf{Kind}.\ (j \to \Omega) \to \Omega \\
\mid \exists_{\mathsf{Kind}} &: \Pi j : \mathsf{Kind}.\ (j \to \Omega) \to \Omega \\
\mid \forall_{\mathsf{Kscm}} &: \Pi z : \mathsf{Kscm}.\ (z \to \Omega) \to \Omega \\
\mid \exists_{\mathsf{Kscm}} &: \Pi z : \mathsf{Kscm}.\ (z \to \Omega) \to \Omega
\end{aligned}$$

Informally, all well-formed computations have types of kind Ω, including singleton types of natural numbers $\mathsf{snat}\ A$ and boolean values $\mathsf{sbool}\ B$, as well as function, tuple, polymorphic and existential types. To improve readability we also define the syntactic sugar

$$\left.\begin{aligned}
A \to B &\equiv \twoheadrightarrow\ A\ B \\
\forall_s X : A.\ B &\equiv \forall_s\ A\ (\lambda X : A.\ B) \\
\exists_s X : A.\ B &\equiv \exists_s\ A\ (\lambda X : A.\ B)
\end{aligned}\right\} \text{where } s \in \{\mathsf{Kind}, \mathsf{Kscm}\}$$

and often drop the sort s when $s = \mathsf{Kind}$; *e.g.* the type void, containing no values, is defined as $\forall \alpha : \Omega.\ \alpha \equiv \forall_{\mathsf{Kind}}\ \Omega\ (\lambda \alpha : \Omega.\ \alpha)$.

Using this syntactic sugar we can give a familiar look to many of the formation rules for λ_H expressions and functional values. Figure 5.6 contains the inference rules for deriving judgments of the form $\Delta; \Gamma \vdash e : A$, which assign type A to the expression e in a context Δ and a type environment Γ defined by

$$(\textit{type env}) \quad \Gamma ::= \cdot \mid \Gamma, x : A$$

We introduce some of the notation used in these rules in the course of the discussion.

Rules E-NAT, E-TRUE, and E-FALSE assign singleton types to numeric and boolean constants.

$$\frac{\Delta \vdash \mathsf{Kind} : \mathsf{Kscm}}{\Delta \vdash \cdot \ ok} \quad \text{(TE-MT)} \qquad \frac{\Delta \vdash \Gamma \ ok}{\Delta; \Gamma \vdash x : \Gamma(x)} \quad \text{(E-VAR)}$$

$$\frac{\Delta \vdash \Gamma \ ok \qquad \Delta \vdash A : \Omega}{\Delta \vdash \Gamma, x{:}A \ ok} \quad \text{(TE-ADD)} \qquad \frac{\Delta \vdash \Gamma \ ok}{\Delta; \Gamma \vdash \overline{n} : \mathsf{snat}\ \widehat{n}} \quad \text{(E-NAT)}$$

$$\frac{\Delta \vdash \Gamma \ ok}{\Delta; \Gamma \vdash \mathsf{tt} : \mathsf{sbool\ true}} \quad \text{(E-TRUE)} \qquad \frac{\Delta \vdash \Gamma \ ok}{\Delta; \Gamma \vdash \mathsf{ff} : \mathsf{sbool\ false}} \quad \text{(E-FALSE)}$$

$$\frac{\Delta \vdash A : \Omega \qquad \Delta; \Gamma, x{:}A \vdash f : A}{\Delta; \Gamma \vdash \mathsf{fix}\ x{:}A.\, f : A} \quad \text{(E-FIX)}$$

$$\frac{\Delta \vdash A : \Omega \qquad \Delta; \Gamma, x{:}A \vdash e : A'}{\Delta; \Gamma \vdash \lambda x{:}A.\, e : A \to A'} \quad \text{(E-FUN)}$$

$$\frac{\Delta; \Gamma \vdash e_1 : A \to A' \qquad \Delta; \Gamma \vdash e_2 : A}{\Delta; \Gamma \vdash e_1\ e_2 : A'} \quad \text{(E-APP)}$$

$$\frac{\Delta \vdash B : s \qquad \Delta, X{:}B; \Gamma \vdash f : A}{\Delta; \Gamma \vdash \Lambda X{:}B.\, f : \forall_s X{:}B.\, A} \ \begin{pmatrix} X \notin \Delta \\ s \neq \mathsf{Ext} \end{pmatrix} \quad \text{(E-TFUN)}$$

$$\frac{\Delta; \Gamma \vdash e : \forall_s X{:}B.\, A' \qquad \Delta \vdash A : B}{\Delta; \Gamma \vdash e[A] : [A/X]A'} \ (s \neq \mathsf{Ext}) \quad \text{(E-TAPP)}$$

$$\frac{\begin{array}{c}\Delta \vdash A : B \qquad \Delta \vdash B : s \\ \Delta; \Gamma \vdash e : [A/X]A'\end{array}}{\Delta; \Gamma \vdash \langle X = A,\ e{:}A' \rangle : \exists_s X{:}B.\, A'} \ (s \neq \mathsf{Ext}) \quad \text{(E-PACK)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e : \exists_s X'{:}B.\, A \qquad \Delta \vdash A' : \Omega \\ \Delta, X{:}B; \Gamma, x{:}[X/X']A \vdash e' : A'\end{array}}{\Delta; \Gamma \vdash \mathsf{open}\ e\ \mathsf{as}\ \langle X,\ x \rangle\ \mathsf{in}\ e' : A'} \ \begin{pmatrix} X \notin \Delta \\ s \neq \mathsf{Ext} \end{pmatrix} \quad \text{(E-OPEN)}$$

$$\frac{\Delta; \Gamma \vdash e : \mathsf{snat}\ A \qquad \Delta; \Gamma \vdash e' : \mathsf{snat}\ A'}{\Delta; \Gamma \vdash e + e' : \mathsf{snat}\ (\mathsf{plus}\ A\ A')} \quad \text{(E-ADD)}$$

$$\frac{\Delta; \Gamma \vdash e : \mathsf{snat}\ A \qquad \Delta; \Gamma \vdash e' : \mathsf{snat}\ A'}{\Delta; \Gamma \vdash e < e' : \mathsf{sbool}\ (\mathsf{lt}\ A\ A')} \quad \text{(E-LT)}$$

Figure 5.6: Static semantics of the computation language λ_H

$$\frac{\textit{for all } i < n \qquad \Delta;\, \Gamma \vdash\ e_i : A_i}{\begin{array}{l}\Delta;\, \Gamma \vdash\ \langle e_0,\, \ldots\, e_{n-1}\rangle \\ \qquad : \nprec\ \widehat{n}\ (\mathsf{nth}\ (A_0 :: \ldots :: A_{n-1} :: \mathsf{nil}))\end{array}} \qquad \text{(E-TUP)}$$

$$\frac{\Delta;\, \Gamma \vdash\ e : \nprec\ A''\ B \qquad \Delta;\, \Gamma \vdash\ e' : \mathsf{snat}\ A' \qquad \Delta \vdash\ A : \mathsf{LT}\ A'\ A''}{\Delta;\, \Gamma \vdash\ \mathsf{sel}[A](e, e') : B\ A'} \qquad \text{(E-SEL)}$$

$$\frac{\begin{array}{ll}\Delta \vdash\ B : \mathsf{Bool} \to \mathsf{Kind} & \Delta;\, \Gamma \vdash\ e : \mathsf{sbool}\ A'' \\ \Delta \vdash\ A : B\ A'' & \Delta, X_1 : B\ \mathsf{true};\, \Gamma \vdash\ e_1 : A' \\ \Delta \vdash\ A' : \Omega & \Delta, X_2 : B\ \mathsf{false};\, \Gamma \vdash\ e_2 : A'\end{array}}{\Delta;\, \Gamma \vdash\ \mathsf{if}[B, A](e,\, X_1.\, e_1,\, X_2.\, e_2) : A'} \qquad \text{(E-IF)}$$

$$\frac{\Delta;\, \Gamma \vdash\ e : A \qquad A =_{\beta\eta\iota} A' \qquad \Delta \vdash\ A' : \Omega}{\Delta;\, \Gamma \vdash\ e : A'} \qquad \text{(E-CONV)}$$

Figure 5.6: Static semantics of the computation language λ_H(contd.)

For instance the constant $\overline{1}$ has type $\mathsf{snat}\ (\mathsf{succ}\ \mathsf{zero})$ in any valid environment. In rule E-NAT we use the meta-function $\widehat{\cdot}$ to map natural numbers $n \in \mathbf{N}$ to their representations as type terms. It is defined inductively by $\widehat{0} = \mathsf{zero}$ and $\widehat{n+1} = \mathsf{succ}\ \widehat{n}$, so $\Delta \vdash\ \widehat{n} : \mathsf{Nat}$ holds for all valid Δ and $n \in \mathbf{N}$.

Singleton types play a central role in reflecting properties of values in the type language, where we can reason about them constructively. For instance rules E-ADD and E-LT use respectively the type terms plus and lt (defined in Section 5.3) to reflect the semantics of the term operations into the type level via singleton types.

However, if we could only assign singleton types to computation terms, in a decidable type system we would only be able to typecheck terminating programs. We regain expressiveness of the computation language using existential types to hide some of the too detailed type information. Thus for example one can define the usual types of all natural numbers and boolean values as

$$\begin{aligned}\mathsf{nat}\ &: \Omega = \exists \alpha : \mathsf{Nat}.\, \mathsf{snat}\ \alpha \\ \mathsf{bool}\ &: \Omega = \exists \alpha : \mathsf{Bool}.\, \mathsf{sbool}\ \alpha\end{aligned}$$

For any term e with singleton type $\mathsf{snat}\ A$ the package $\langle \alpha = A,\, e : \mathsf{snat}\ \alpha \rangle$ has type nat. Since in a type-erasure semantics of λ_H all types and operations on them are erased, there is no runtime

overhead for the packaging. For each $n \in \mathbf{N}$ there is a value of this type denoted by $\widehat{\overline{n}} \equiv \langle \alpha = \widehat{n},\, \overline{n} : \mathsf{snat}\ \alpha \rangle$. Operations on terms of type nat are derived from operations on terms of singleton types of the form $\mathsf{snat}\ A$; for example an addition function of type $\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$ is defined as the expression

$$\begin{aligned}\mathsf{add} = \lambda \mathsf{x}_1 : \mathsf{nat}.\, \lambda \mathsf{x}_2 : \mathsf{nat}.\\ \mathsf{open}\ \mathsf{x}_1\ \mathsf{as}\ \langle \alpha_1,\, \mathsf{x}_1' \rangle\ \mathsf{in}\ \mathsf{open}\ \mathsf{x}_2\ \mathsf{as}\ \langle \alpha_2,\, \mathsf{x}_2' \rangle\ \mathsf{in}\\ \langle \alpha = \mathsf{plus}\ \alpha_1\ \alpha_2,\, \mathsf{x}_1' + \mathsf{x}_2' : \mathsf{snat}\ \alpha \rangle\end{aligned}$$

Rule E-TUP assigns to a tuple a type of the form $\Join\ A\ B$, in which the $\Join$ constructor is applied to a type A representing the tuple size, and a function B mapping offsets to the types of the tuple components. This function is defined in terms of operations on lists of types:

$$\begin{aligned}&\mathsf{Inductive\ List} : \mathsf{Kind} := \mathsf{nil} \quad : \mathsf{List}\\ &\qquad\qquad\qquad\qquad\quad \mid \mathsf{cons} : \Omega \to \mathsf{List} \to \mathsf{List}\\ \\ &\mathsf{nth} : \mathsf{List} \to \mathsf{Nat} \to \Omega\\ &\mathsf{nth\ nil} \qquad\qquad = \lambda \alpha : \mathsf{Nat}.\, \mathsf{void}\\ &\mathsf{nth}\ (\mathsf{cons}\ \alpha_1\ \alpha_2) = \lambda \alpha : \mathsf{Nat}.\, \mathsf{ifez}\ \alpha\ \Omega\ \alpha_1\ (\mathsf{nth}\ \alpha_2)\end{aligned}$$

Thus $\mathsf{nth\ L}\ \widehat{n}$ reduces to the n-th element of the list L when n is less than the length of L, and to void otherwise. We also use the infix form $A :: A' \equiv \mathsf{cons}\ A\ A'$. The type of pairs is derived: $A \times A' \equiv \Join\ \widehat{2}\ (\mathsf{nth}\ (A :: A' :: \mathsf{nil}))$. Thus for instance $\cdot ; \cdot \vdash \langle \overline{42}, \overline{7} \rangle : \mathsf{snat}\ \widehat{42} \times \mathsf{snat}\ \widehat{7}$ is a valid judgment.

The rules for selection and testing for the less-than relation refer to the kind term LT with kind schema $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Kind}$. Intuitively, LT represents a binary relation on kind Nat, so $\mathsf{LT}\ \widehat{m}\ \widehat{n}$ is the kind of type terms representing proofs of $m < n$. LT can be thought of as the parameterized inductive kind of proofs constructed from instances of the axioms $\forall n \in \mathbf{N}.\, 0 < n{+}1$ and $\forall m, n \in \mathbf{N}.\, m < n \supset m{+}1 < n{+}1$:

$$\begin{aligned}&\mathsf{Inductive\ LT} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Kind}\\ &\quad := \mathsf{ltzs} : \Pi \alpha : \mathsf{Nat}.\, \mathsf{LT\ zero}\ (\mathsf{succ}\ \alpha)\\ &\qquad \mid \mathsf{ltss} : \Pi \alpha : \mathsf{Nat}.\, \Pi \alpha' : \mathsf{Nat}.\, \mathsf{LT}\ \alpha\ \alpha' \to \mathsf{LT}\ (\mathsf{succ}\ \alpha)\ (\mathsf{succ}\ \alpha')\end{aligned}$$

In our type language we allow inductive kinds of kind scheme Kind only. Thus we actually need

to use a Church encoding of LT (see [SSTP01] for the encoding).

In the component selection construct $\mathsf{sel}[A](e, e')$ the type A represents a *proof* that the value of the subscript e' is less than the size of the tuple e. In rule E-SEL this condition is expressed as an application of the type term LT. Due to the consistency of the logic represented in the type language, only the existence and not the structure of the proof object A is important. Since its existence is ensured statically in a well-formed expression, A would be eliminated in a type-erasure semantics.

The branching construct $\mathsf{if}[B, A](e,\ X_1.\, e_1,\ X_2.\, e_2)$ takes a type term A representing a proof of the proposition encoded as either $B\ \mathsf{true}$ or $B\ \mathsf{false}$, depending on the value of e. The proof is passed to the appropriate branch in its bound type variable (X_1 or X_2). The correspondence between the value of e and the kind of A is again established through a singleton type.

5.5.2 Example: bound check elimination

A simple example of the generation, propagation, and use of proofs in λ_H is a function which computes the sum of the components of any vector of naturals. Let us first introduce some auxiliary types and functions. The type assigned to a homogeneous tuple (vector) of n terms of type A is $\beta\eta\iota$-convertible to the form $\mathsf{vec}\ \widehat{n}\ A$ for

$$\begin{aligned}&\mathsf{vec} : \mathsf{Nat} \to \Omega \to \Omega\\ &\mathsf{vec} = \lambda\alpha{:}\mathsf{Nat}.\, \lambda\alpha'{:}\Omega.\, \divideontimes\ \alpha\ (\mathsf{nth}\ (\mathsf{repeat}\ \alpha\ \alpha'))\end{aligned}$$

where

$$\begin{aligned}&\mathsf{repeat} : \mathsf{Nat} \to \Omega \to \mathsf{List}\\ &\mathsf{repeat\ zero} \quad\ = \lambda\alpha'{:}\Omega.\, \mathsf{nil}\\ &\mathsf{repeat}\ (\mathsf{succ}\ \alpha) = \lambda\alpha'{:}\Omega.\, \alpha'{::}(\mathsf{repeat}\ \alpha)\ \alpha'\end{aligned}$$

Then we can define a term which sums the elements of a vector with a given length as follows:

$$
\begin{array}{l}
\mathsf{sumVec} : \forall\alpha{:}\mathsf{Nat}.\, \mathsf{snat}\ \alpha \to \mathsf{vec}\ \alpha\ \mathsf{nat} \to \mathsf{nat} \\
\quad \equiv \Lambda\alpha{:}\mathsf{Nat}.\, \lambda\mathsf{n}{:}\mathsf{snat}\ \alpha.\, \lambda\mathsf{v}{:}\mathsf{vec}\ \alpha\ \mathsf{nat}. \\
\qquad (\mathsf{fix}\ \mathsf{loop}{:}\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}. \\
\qquad\quad \lambda\mathsf{i}{:}\mathsf{nat}.\, \lambda\mathsf{sum}{:}\mathsf{nat}. \\
\qquad\qquad \mathsf{open}\ \mathsf{i}\ \mathsf{as}\ \langle\alpha', \mathsf{i}'\rangle\ \mathsf{in} \\
\qquad\qquad\ \mathsf{if}[\mathsf{LTOrTrue}\ \alpha'\ \alpha, \mathsf{ltPrf}\ \alpha'\ \alpha] \\
\qquad\qquad\quad (\mathsf{i}' < \mathsf{n}, \\
\qquad\qquad\qquad \alpha_1.\, \mathsf{loop}\ (\mathsf{add}\ \mathsf{i}\ \widehat{1})\ (\mathsf{add}\ \mathsf{sum}\ (\mathsf{sel}[\alpha_1](\mathsf{v}, \mathsf{i}'))), \\
\qquad\qquad\qquad \alpha_2\,.\,\mathsf{sum})) \widehat{0}\, \widehat{0}
\end{array}
$$

where

$$
\begin{array}{l}
\mathsf{LTOrTrue}\ :\ \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Bool} \to \mathsf{Kind} \\
\mathsf{LTOrTrue} = \lambda\alpha_1{:}\mathsf{Nat}.\, \lambda\alpha_2{:}\mathsf{Nat}.\, \lambda\alpha{:}\mathsf{Bool}.\, \mathsf{Cond}\ \alpha\ (\mathsf{LT}\ \alpha_1\ \alpha_2)\ \mathsf{True}
\end{array}
$$

The comparison $\mathsf{i}' < \mathsf{n}$, used in this example as a loop termination test, checks whether the index i' is smaller than the vector size n. If it is, the adequacy of the type term lt with respect to the less-than relation ensures that the type term $\mathsf{ltPrf}\ \alpha'\ \alpha$ represents a proof of the corresponding proposition at the type level, namely $\mathsf{LT}\ \alpha'\ \alpha$. This proof is then bound to α_1 in the first branch of the if, and the sel construct uses it to verify that the i'-th element of v exists, thus avoiding a second test. The type safety of λ_H (Theorem 5.5.1) guarantees that implementations of sel need not check the subscript at runtime. Since the proof α_2 is ignored in the "else" branch, $\mathsf{ltPrf}\ \alpha'\ \alpha$ is defined to reduce to the trivial proof of True when the value of i' is not less than that of n.

As an aside, the usual vector type, which keeps the length packaged with the content, is then:

$$\mathsf{vector}\ : \Omega \to \Omega = \lambda\alpha{:}\Omega.\, \exists\alpha'{:}\mathsf{Nat}.\, \mathsf{snat}\ \alpha' \times \mathsf{vec}\ \alpha'\ \alpha.$$

We won't give here the proof of type-safety for this sample computation language λ_H. The interested reader may refer to [SSTP01] for the details. The proof does not contain any new subtleties.

Theorem 5.5.1 (Safety of λ_H) *If $\cdot;\cdot \vdash\ e : A$, then either e is a value or there exists an e' such that*

$e \mapsto e'$ *and* $\cdot;\cdot \vdash e' : A$.

5.6 Summary and related work

To sum up, an intermediate language is now constructed by combining our type language λ^i_{CC} with a particular computation language. The type system for the particular computation language is created by giving the inductive definition for the base kind Ω. For example, the computation language here was λ_H with the base kind defined at the beginning of Section 5.5.1.

Our type language is a variant of the calculus of constructions [CH88] extended with inductive definitions (with both small and large elimination) [Pau93, Wer94]. We support η-reduction in our language while the official Coq system does not. The proofs for the properties of λ^i_{CC} are adapted from Geuvers [Geu93] and Werner [Wer94]; the main difference is that our language has kind-schema variables and a new product formation rule $(\mathsf{Ext}, \mathsf{Kind})$ which are not in Werner's system.

The Coq proof assistant provides support for extracting programs from proofs [Pau93]. It separates propositions and sets into two distinct universes Prop and Set. We do not distinguish between them because we are not aiming to extract programs from our proofs, instead, we are using proofs as specifications for our computation terms.

Xi and Pfenning's DML [XP99] is the first language that nicely combines dependent types with programs that may involve effects. Our ideas of using singleton types and lifting the level of the proof language are directly inspired by their work. Xi's system, however, does not support arbitrary propositions and explicit proofs. It also does not define the Ω kind as an inductive definition so it is unclear how it interacts with intensional type analysis [TSS00] and how it preserves proofs during compilation.

We have discussed the relationship between our work and those on PCC, typed assembly languages, and intensional type analysis in Section 5.1. Inductive definitions subsume and generalize earlier systems on intensional type analysis [HM95, CWM99, TSS00]; the type-analysis construct in the computation language can be eliminated using the technique proposed by Crary *et al.* [CWM98].

Crary and Vanderwaart [CV01] recently proposed a system called LTT which also aims at adding explicit proofs to typed intermediate languages. LTT uses Linear LF [CP96] as its proof

language. It shares some similarities with our system in that both are using singleton types [XP99] to circumvent the problems of dependent types. However, since LF does not have inductive definitions and the Elim construct, it is unclear how LTT can support intensional type analysis and type-level primitive recursive functions [CW00].

Chapter 6

Implementation

In this chapter we discuss the implementation details of the type system for certified runtime type analysis. We implemented the type system in an experimental version of the Standard ML of New Jersey compiler [AM91]. The motivation for the implementation was two-fold. First, we wanted to create an infrastructure that can be used for writing certified applications. Second we wanted to measure the overhead of implementing such an expressive type system in an industrial-strength compiler.

The current SML/NJ compiler (v110.34) is a type-preserving compiler; in fact, types are propagated through all optimizations and right till code generation. The intermediate language of the compiler is based on the predicative F_ω calculus [Sha97b]. The type language, by itself, is similar to the simply typed lambda calculus.

Our implementation uses the frontend and the code generator from the existing compiler. The type preserving phase is new, and uses the certifying type system described in Chapter 5. We implemented this system since it integrates runtime type analysis and a logical system inside a single framework. We implemented the new type system and the associated type manipulation operations, and then re-wrote all the optimizations to use this new type system. Thus the type-preserving phases in both the compilers are the same: we do inlining, specialization, loop optimizations, lay down efficient data representations, *etc*. Figure 6.1 illustrates the design of the two compilers.

We have not used the new type system to prove safety properties of programs; however, the implementation maintains all the information that is needed to write certified applications. In the old implementation the base kind Ω (the kind for the type of terms) was an in-built primitive kind. As shown in Chapter 5, the new type system provides a mechanism for defining kinds inductively.

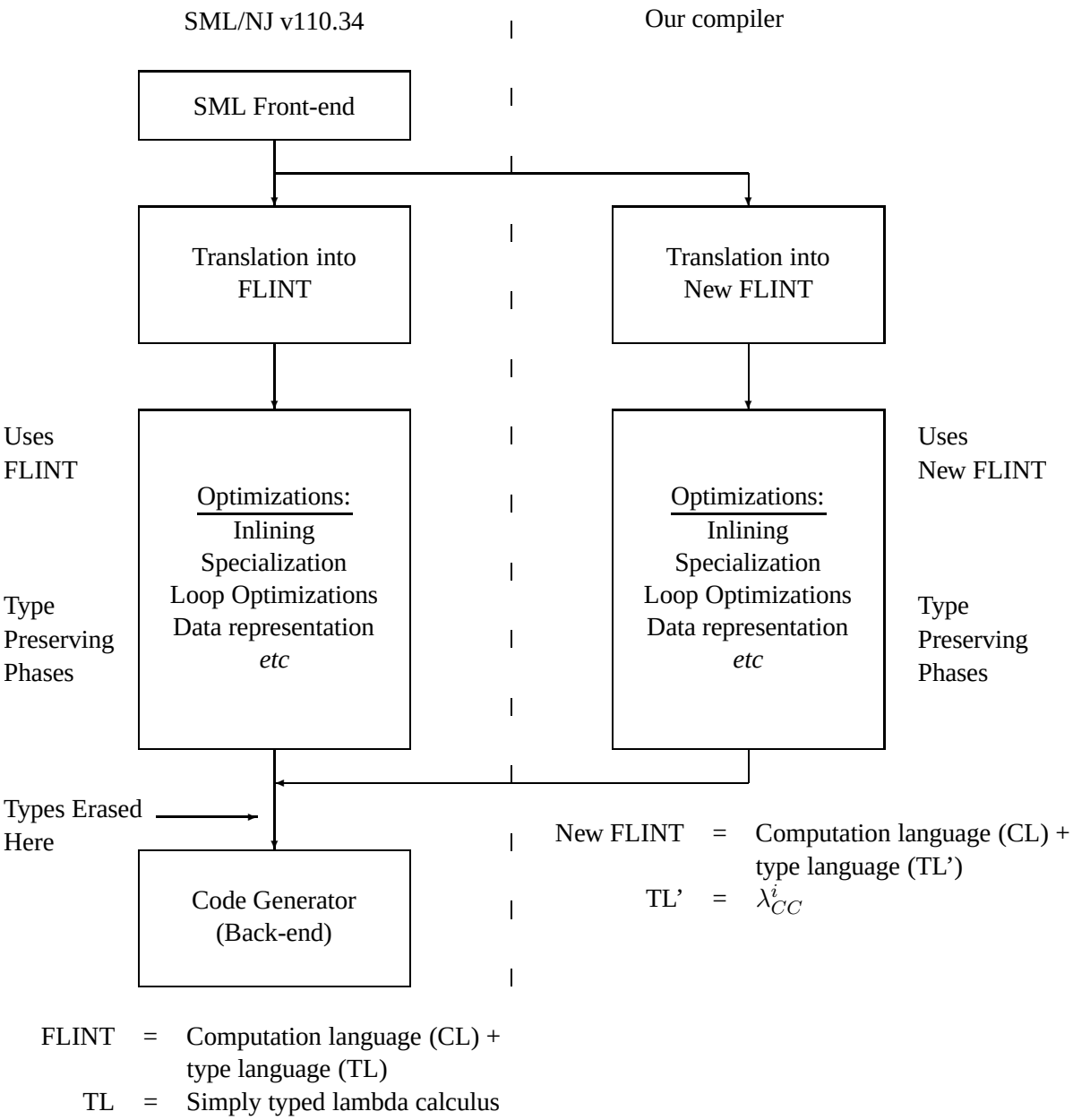


Figure 6.1: Comparing the implementations

Benchmark	No of lines	Average type size	Type size distribution			
			$> 10^2$	$> 10^3$	$> 10^4$	$> 10^5$
Mlyacc	6030	36	510	56	_	_
Vliw	3679	73	874	197	_	_
CML	6152	81	1359	34	27	_

Figure 6.2: Size of types generated during compilation

The base kind Ω is now an inductively defined kind (see Section 6.4); type constructors such as tuples, vectors, arrays, function types, *etc* are members of this inductive definition. Furthermore, each of these types carry more information. For example, a tuple type not only carries the type of each component, but also the length of the tuple. Therefore, the new compiler creates the scaffolding required for certifying programs; thus, we can measure the overhead introduced by the new type system.

How does a type system impact the compilation strategy ? To get an idea we took some measurements of the amount of type information generated by the Standard ML of New Jersey compiler during the compilation process. Figure 6.2 shows the figures on three large benchmarks. Mlyacc is a parser generator for Standard ML, vliw is an instruction scheduler for a vliw compiler, and CML is an implementation of Concurrent-ML [Rep91]. The amount of type information is measured in terms of the number of nodes in a type. Essentially, if we consider a type as a tree, then this table counts the number of nodes in the tree. For example, a primitive type like int is of size one, a pair $\mathsf{int} \times \mathsf{int}$ is of size two, a type $(\mathsf{int} \times \mathsf{int}) \rightarrow \mathsf{int}$ is of size three, and so on. The sizes are calculated without taking any sharing (between types) into consideration. The third column in Figure 6.2 shows the average number of nodes in a type. For example, during the compilation of mlyacc a type on the average has 36 nodes. The last column shows the spread in the size of the types: during the compilation of mlyacc we generate 510 types with each of them having more than a hundred nodes, and 56 types with each of them having more than a thousand nodes.

This table shows that the overhead incurred from a type system is surprisingly large. For example, on CML the compilation generates a good number of types with more than ten thousand nodes each. Since the type system in the compiler is similar to the simply typed lambda calculus, these figures reflect the overhead of using the simplest system in the lambda-cube. Therefore, one of the key questions is: will the type information that must be maintained for the most expressive system (in the lambda-cube) overwhelm the compiler ?

6.1 Implementing a pure type system

The implementation, like the formalization, uses the PTS specification of the type system. This allows us to use the same datatype for types, kinds, and kind schemas: therefore, it provides a tremendous economy in the number of constructs that need to be implemented. The datatype pterm used for representing the PTS is shown below:

```
datatype sort
       = Kind
       | Kscm
       | Ext

datatype pterm
       = Var of int * sort                          (* variables *)
       | Srt of sort                                (* sorts *)
       | Pi of pterm * pterm                        (* product *)
       | Fn of pterm * pterm                        (* abstraction *)
       | App of pterm * pterm                       (* application *)
       | Ind of pterm * int * pterm list            (* inductive definition *)
       | Con of pterm * int * int * pterm list      (* constructor of inductive definition *)
       | Fix of int list * pterm * int * pterm list (* fixpoint *)
       | Case of pterm * pterm * pterm list         (* case construct *)
       | Seq of pterm list                          (* record *)
       | Proj of pterm * int                        (* projection *)
       | Clos of pterm * int * int * env            (* closure *)


datatype env
       = Emp                                        (* empty environment *)
       | Sub of (int * pterm) * env                 (* environment with a substitution *)
```

We use DeBruijn indices [De 80] for variables. Thus a term of the form $\lambda X : A.\, \lambda Y : X. \ldots Y \ldots X \ldots$ is represented as $\lambda\_ : A.\, \lambda\_ : \#1. \ldots \#1 \ldots \#2 \ldots$ where the variable is denoted by its lexical depth. This is useful because two α-equivalent terms now have the same representation. The variables contain another bit of information. Since we are using a PTS

representation, there is no syntactic distinction between types, kinds, and schemas. But for type-checking, this information is often useful: the extra bit memoizes this information. Our binding constructs (Fn and Pi) now only need to store the type of the bound variable. For example, $\lambda X : A.\, B$ is represented as $Fn(p_1, p_2)$ where p_1, p_2 are the representations for A and B respectively.

The mechanism for inductive definition is more general than that presented in Chapter 5. The implementation allows mutually recursive, parametrized inductive definitions. The first sub-term contains the body of the inductive definitions, the integer says which definition we are interested in, and the list at the end represents the arguments. Consider the inductive definitions shown below. We first define the natural numbers and then define an integer list that keeps track of its length. The empty list has zero length. When we cons an integer to a list, we increase its length by one:

Inductive (Nat:Kind)
= zero : Nat
| succ : Nat → Nat

Inductive (ListNat:Nat → Kind)
= nil : ListNat(zero)
| cons : ΠX : Nat. int → ListNat(X) → ListNat(succ(X))

Suppose that we wanted to represent the inductive definition of ListNat. The arrow type → should be represented as a product type, but we will use the arrow type wherever possible to make the presentation clearer. We first build the pseudoterm p representing the definition:

p = Fn (Seq [Nat$_{pts}$ → $Kind$],
Seq [
Seq[App(#1,zero$_{pts}$), Pi(Nat$_{pts}$, int → App(#2, #1) → App(#2, succ$_{pts}$(#1))]
]
)

An inductive definition binds a set of names; we use the Fn construct for this. As is the case with ordinary functions, the Fn construct carries only the type of the bound variables. Since we may have mutually inductive definitions, we use the Seq construct that lets us bind a list of variables. In the example above, the inductive definition introduces a variable of type Nat → Kind; therefore, Fn binds a Seq [Nat$_{pts}$ → $Kind$]. We use Nat$_{pts}$ to denote the PTS representation of the

inductive definition Nat. The body of Fn contains the definition of the constructors. The body is a sequence where every member contains the constructors (as a sequence) for one particular inductive definition. The variables introduced by the definition (in this case ListNat) are denoted by DeBruijn indices. We use zero$_{pts}$ and succ$_{pts}$ to denote the PTS representation of the constructors zero and succ An integer list of length one has the type ListNat(succ(zero)). This would be represented as: Ind(p, 1, [succ$_{pts}$(zero$_{pts}$)]).

The representation of constructors contains the representation of the inductive definition. For example, the constructor nil is represented as Con(p, 1, 1, []). This says that nil is the first constructor in the first set of definitions in p, and the set of arguments for this constructor is empty.

In the presentation in Chapter 5 we had a built-in primitive recursion operator that we called Elim. In our implementation, we use two other built-in operators, a $Case$ construct and a recursion construct (Fix) to simulate this. The formation rules and the reduction rules for these constructs is entirely standard. This approach was formalized and proved equivalent to the primitive recursive approach in [Gim98, HPM$^+$00]. We used the same formalization in our implementation.

$$\begin{aligned} & Case(p, p_1, \vec{p'}) \rightsquigarrow p'_j\ p''_1 \ldots p''_n \\ \text{where} \quad & p_1 = Con(p_3, i, j, [p''_1, \ldots, p''_n]) \end{aligned}$$

The PTS representation of the case construct is explicitly typed. The first sub-term p gives the type of the case construct; the next sub-term p_1 is the term being analyzed (which must have an inductive type); and the list $\vec{p'}$ is the set of branches for the case statement. Every branch must bind the arguments of the corresponding constructor; these arguments, $p''_1 \ldots p''_n$, are passed in during the reduction.

Our recursive construct can include several mutually recursive definitions, but the syntax includes special restrictions to prevent non-normalizing terms. One of the arguments of the Fix construct must have an inductive type. This is the argument that we will recurse on. The body of Fix must start with a case analysis of this inductive argument. As pointed before, the case analysis binds the arguments of the constructors to variables. Recursive calls are allowed only on these variables. Intuitively, the recursion can occur only on the subterms of an inductively defined term. Therefore, the syntax of Fix must specify which argument is being recursed on.

The abstract syntax for a set of mutually recursive definitions is the following:

$$Fix_j(f_1/i_1 : A_1 = B_1 \qquad \ldots \qquad f_n/i_n : A_n = B_n)$$

The body of f_k is B_k and it has the type A_k. In the body B_k, we will recurse over the ${i_k}^{th}$ argument. We are interested in the j^{th} definition. We will use $Fix_j(F)$ to denote this definition. At every occurrence this fixpoint must be applied to i_j arguments, where the ${i_j}^{th}$ argument must have an inductive type. The reduction rule (shown below) is standard; however, the reduction is applied only if ${A'}_{i_j}$ (the ${i_j}^{th}$ argument) is syntactically a constructor. This restriction is needed to ensure strong normalization, and corresponds to the reduction for primitive recursive operators.

$$Fix_j(F)\ A'_1 \ldots {A'}_{i_j} \rightsquigarrow ([(Fix_k(F)/f_k)_{k=1\ldots n}]B_j)\ A'_1 \ldots {A'}_{i_j}$$

The PTS representation of this fixpoint occurrence is shown below:

$$\begin{array}{ll} & Fix(\vec{i}, p, j, \vec{p'}) \\ \text{where} & \vec{i} = [i_1, \ldots, i_n] \\ & p = Fn(Seq[A_{1pts}, \ldots, A_{npts}], Seq[B_{1pts}, \ldots, B_{npts}]) \\ & \vec{p'} = [A'_{1pts}, \ldots, {A'}_{i_j - 1_{pts}}] \end{array}$$

As before we use a Fn construct to bind the sequence of variables $f_1 \ldots f_n$ in the bodies B_{ipts}. The list of integers $\vec{i}$ specifies the argument in each recursive definition that is being recursed on. The list of terms $\vec{p'}$ carries all the arguments previous to the one that is being recursed on.

The Seq and $Proj$ constructs are mainly an implementation aid. The Seq construct is used for binding a list of pterms, for example the set of constructors in an inductive definition. By using the Seq construct we ensure through our hash-consing scheme (Section 6.3) that there is only a single copy of the list in memory. If we use an explicit list of pterms then each component of the list would be shared; however, we would still have multiple copies of the list. Moreover, the Seq and $Proj$ constructs allow us to handle a list of terms without any meta-level machinery. Since environments can also be modelled as a list of bindings, these constructs are also used for sharing environments.

6.2 Rationale for design decisions

The main motivations in designing the representation of PTS terms were to provide fast type manipulation, and to make type-checking convenient. These were reflected mainly in the representation of inductive definitions, and elimination forms for these inductive definitions. In this section we will explain why we chose these representations with some simple examples.

Constructors: During type reductions constructors are mainly used for analysis in the $Case$ construct. Consider the reduction of a case statement: $Case(p_1, p_2, \vec{p'})$. We need to figure out whether p_2 is a constructor, and if so, the arguments of the constructor. Since our representation for a constructor carries all its arguments, we have the invariant that a term is a constructor if and only if its representation is of the form $Con(p_1, i, j, \vec{p'})$. On the other hand if the constructor was explicitly applied to arguments, we would have to deconstruct a sequence of applications, and determine whether the term at the head was a constructor. Moreover, our representation also makes it easy to recover the arguments.

In the representation for a constructor $Con(p_1, i, j, \vec{p'})$, the term p_1 contains the definition of all the constructors in the corresponding inductive definition. Even though this seems, at first glance, to be an overkill, it does not incur a significant overhead. First, our hash-consing mechanism (Section 6.3) ensures that there is only a single copy of p_1 in memory; all occurrences of the constructors of an inductive definition share the same copy.

$$\frac{\Delta \vdash I : A \textit{ where } I = \mathsf{Ind}(X:A)\{\vec{C}\}}{\Delta \vdash \mathsf{Ctor}\,(i, I) : [I/X]C_i}$$

Second, consider the typing rule for a constructor shown above (we are considering just a simple inductive definition). The type of a constructor is obtained by substituting the bound variable by the inductive definition $[I/X]C_i$. In the general case of a mutually inductive definition, each of the bound variables will have to be substituted. Our representation for constructors makes this easy since p_1 encapsulates the entire inductive definition.

Inductive definition: Parametrized inductive definitions are represented as $Ind(p_1, i, \vec{p'})$, where $\vec{p'}$ are the values of the parameters. Consider the parametrized definition of lists that we saw before:

$$\begin{array}{lll} \textsf{Inductive} & \text{(ListNat:Nat} \rightarrow \textsf{Kind)} & \\ & = \quad \text{nil} \quad : \text{ListNat(zero)} & \\ & \mid \quad \text{cons} \quad : \Pi X : \text{Nat.}\, \textsf{int} \rightarrow \text{ListNat}(X) \rightarrow \text{ListNat}(\text{succ}(X)) \end{array}$$

In the type of the constructors, ListNat is fully applied to arguments. Since an inductive definition occurs with its parameters instantiated, we again maintain an invariant similar to the constructors: a term is an inductive definition if and only if it has the form $Ind(p_1, i, \vec{p'})$. Moreover, when we check the positivity condition of the constructors, we have to check that the arguments of a parametrized definition satisfy certain constraints. Our representation makes it easy to extract the arguments.

Fixpoint definition: We use a similar mechanism for our fixpoint definitions: namely, the representation includes all the arguments till the one that is being recursed on. For example in $Fix(\vec{i}, p_1, j, \vec{p'})$, the list $\vec{p'}$ includes the first $i_j - 1$ arguments. During type reductions, the fixpoint (as shown before) is unfolded only if the i_j^{th} argument is a constructor. During type-checking, we have to check that this argument has an inductive type. Our representation makes it easy to access this argument. The term p_1 uses the standard encoding of fixpoints as functions.

6.3 Implementation Details

Implementing a type system in an industrial-strength compiler is a very difficult problem; a naive implementation can easily add an exponential overhead to the compilation time. Fortunately, previous researchers have implemented scalable implementations of type-preserving production compilers [SLM98]. We used many of the excellent ideas in [SLM98, Nad94] to make our implementation efficient. We kept the following criteria in mind: compact space usage, fast equality operation, and efficient type reductions.

Compact space usage: As we showed in Figure 6.2 large types are ubiquitous in real-world applications. This happens specially in highly-modularized programs because a module interface gets exposed through multiple access paths. As a result the amount of type information that needs to be manipulated multiplies rapidly; however, this type information also has a lot of redundancy. It is crucial that an implementation exploit this redundancy in representing types efficiently. Therefore we *hash-cons* all PTS terms into a global hash table. Our hash-consing scheme guarantees that all

PTS terms use the most compact dag representation. To protect against space leaks, we use *weak pointers* for every hash entry.

Fast type equality operation: Our hash-consing scheme also enables a very fast type equality operation. Since we use DeBruijn indices for variables, α-equivalent terms have the same representation. Hash-consing ensures that all of these terms share the same representation in memory. Moreover, for types in normal form, we can test for equality by using pointer equality. If the types are not in normal form, then we reduce the types to weak-head normal form and check the heads for equality. If the heads are equal, we continue recursively on the sub-terms. In practice, this leads to fast equality tests.

Efficient type reductions: We use a combination of lazy reductions and memoization to make type reductions efficient. For lazy reductions, we add a form of *suspension terms* [NW90, Nad94, SLM98] to our PTS calculus through the $Clos$ construct. Intuitively, these suspension terms are a form of closures: they are a pair of a type and an environment containing the values of the free variables. Reductions change the nesting depth of PTS terms which changes the DeBruijn indices of variables. Therefore, the environment formed during a reduction must remember both the required substitutions and the change in DeBruijn indices. Accordingly, the environments are represented as a triple (i, j, env) where the first index i indicates the current embedding level of the term, the second index j indicates the embedding level after the reduction, and the env is a mapping between variables (DeBruijn indices) to terms.

Figure 6.3 shows how we use these suspension terms. Since the sort of a variable plays no role in the reductions, we represent variables only by their DeBruijn index. We use $Var(1)$ to denote the innermost lambda bound variable. During a β-reduction, instead of performing the substitution directly, we create a suspension term. The environment represents the following: the term p_2 that was originally in the scope of 1 abstraction is now under none; p_3 originally under the scope of 0 abstractions is to be substituted for the first free variable in p_2. In the other cases we propagate the suspensions through to the subterms. This allows us to do the substitution lazily; we query the environment only when we encounter a variable during type manipulations. In the case of a free variable, we simply adjust its DeBruijn index; in the case of a bound variable, we use the value from the environment.

We memoize type reductions to ensure that the same reduction is not repeated. For example,

$$
\begin{array}{ll}
App(Fn(p_1,p_2),p_3) & \leadsto Clos(p_2,(1,0,Sub((p_3,0),Emp))) \quad (\beta-\text{redex}) \\
Clos(Var(n),(i,j,env)) & \leadsto Var(n-i+j) \qquad n>i \quad \text{(free variable)} \\
Clos(Var(n),(i,j,env)) & \leadsto Clos(p_1,(0,j-j',Emp)) \\
 & \quad n^{th}\text{element of } env=(j',p_1) \quad \text{(bound variable)} \\
Clos(Fn(p_1,p_2),(i,j,env)) & \leadsto Fn(pr^{env}_{i,j}(p_1),ext^{env}_{i,j}(p_2)) \\
Clos(Pi(p_1,p_2),(i,j,env)) & \leadsto Pi(pr^{env}_{i,j}(p_1),ext^{env}_{i,j}(p_2)) \\
Clos(App(p_1,p_2),(i,j,env)) & \leadsto App(pr^{env}_{i,j}(p_1),pr^{env}_{i,j}(p_2)) \\
Clos(Ind(p_1,i,\vec{p'}),(i,j,env)) & \leadsto Ind(pr^{env}_{i,j}(p_1),i,\overrightarrow{pr^{env}_{i,j}(p')}) \\
Clos(Con(p_1,i,j,\vec{p'}),(i,j,env)) & \leadsto Con(pr^{env}_{i,j}(p_1),i,j,\overrightarrow{pr^{env}_{i,j}(p')}) \\
Clos(Fix(\vec{i},p_1,j,\vec{p'}),(i,j,env)) & \leadsto Fix(\vec{i},pr^{env}_{i,j}(p_1),j,\overrightarrow{pr^{env}_{i,j}(p')}) \\
Clos(Case(p_1,p_2,\vec{p'}),(i,j,env)) & \leadsto Case(pr^{env}_{i,j}(p_1),pr^{env}_{i,j}(p_2),\overrightarrow{pr^{env}_{i,j}(p')}) \\
Clos(Seq(\vec{p'}),(i,j,env)) & \leadsto Seq(\overrightarrow{pr^{env}_{i,j}(p')}) \\
Clos(Proj(p_1,k),(i,j,env)) & \leadsto Proj(pr^{env}_{i,j}(p_1),k) \\
Clos(Srt(s),\_) & \leadsto Srt(s)
\end{array}
$$

where

$$
\begin{array}{rcl}
pr^{env}_{i,j}(p) & \equiv & Clos(p,(i,j,env)) \\
ext^{env}_{i,j}(p) & \equiv & Clos(p,(i+1,j+1,Sub((j,Var(1)),env)))
\end{array}
$$

Figure 6.3: Using suspension terms for lazy reduction

suppose we have the term $p = App(Fn(p_1, p_2), p_3)$. Suppose the term p gets reduced to the term $p' = Clos(p_2, (1, 0, Sub((p_3, 0), Emp)))$. After the reduction we update the hash entry for p to contain the term p'. Any future table lookup (during future creations) for the term p will now return the term p'.

The hash-table entry for every term stores additional information. We memoize the set of free variables, whether a term is in normal form, and the universe that it resides in (whether it is a type, kind, or kind schema). The set of free variables is useful in applying substitutions, the normal form flag is useful in equality testing, and the universe is useful in checking inductive definitions.

6.4 Representing the base kind

We showed in Chapter 5 that the key issue in using our type system for a compiler intermediate language was to define the base kind Ω. Correspondingly we must also define the types that are members of Ω such as primitive types, function types, polymorphic types, *etc*. We also showed that to support runtime type analysis, the base kind must be defined inductively. In this section, we show the definition of Ω used in the implementation.

$$
\begin{array}{lll}
\textsf{Inductive} & \textsf{Nat} : \textsf{Kind} & \\
& := \textsf{zero} \quad : \textsf{Nat} & \\
& \mid \quad \textsf{succ} \quad : \textsf{Nat} \rightarrow \textsf{Nat} & \\
\textsf{Inductive} & \Omega : \textsf{Kind} & \\
& := \textsf{snat} \quad : \textsf{Nat} \rightarrow \Omega & \text{(* singleton int *)} \\
& \mid \quad +\!\!+ \quad : \Omega List \rightarrow \Omega & \text{(* sum constructor *)} \\
& \mid \quad \mu\mu \quad : \Pi j : \textsf{Kind}.\ (j \rightarrow j) \rightarrow (j \rightarrow \Omega) \rightarrow \Omega & \text{(* recursive constructor *)} \\
& \mid \quad \forall \quad : \Pi j : \textsf{Kind}.\ (j \rightarrow \Omega) \rightarrow \Omega & \text{(* polymorphic constructor *)} \\
& \mid \quad \exists \quad : \Pi j : \textsf{Kind}.\ (j \rightarrow \Omega) \rightarrow \Omega & \text{(* existential constructor *)} \\
& \mid \quad \twoheadrightarrow \quad : \Omega List \rightarrow \Omega List \rightarrow \Omega & \text{(* function/functor constructor *)} \\
& \mid \quad \ltimes \quad : \textsf{Nat} \rightarrow \Omega List \rightarrow \Omega & \text{(* tuple/structure constructor *)} \\
& \mid \quad \textsf{cont} \quad : \Omega List \rightarrow \Omega & \text{(* continuation type *)} \\
& \mid \quad \textsf{box} \quad : \Omega \rightarrow \Omega & \text{(* box type *)} \\
& \mid \quad \textsf{azero} \quad : \Omega & \text{(* arity zero primitives *)} \\
& \mid \quad \textsf{aone} \quad : \Omega \rightarrow \Omega & \text{(* arity one primitives *)} \\
\textsf{Inductive} & \Omega List : \textsf{Kind} & \\
& := \textsf{nil} \quad : \Omega List & \\
& \mid \quad \textsf{cons} \quad : \Omega \rightarrow \Omega List \rightarrow \Omega List &
\end{array}
$$

The base kind uses the definition of natural numbers shown above. The kind Ω is itself a mutually recursive definition. We use $\Omega List$ to encode a list of types. Our function type constructor works on multiple arguments and return values; therefore it takes arguments of kind $\Omega List$. Our tuple type constructor carries the length of the tuple. The definition of the polymorphic and existential constructors follows from Chapter 3. The box type constructor is used for data representation and the cont type constructor is used during CPS conversion. The azero and aone are used for primitive types. Arity zero primitive types include int31 (31 bit integers), int32, real, string, void, and the primitive exception. Arity one primitive types include the list constructor, array, ref, vector, exception tag, and the primitive box constructor.

6.4.1 Example: Flattening arguments

Standard ML provides single argument/return-value functions; multiple values are simulated by using tuples. Since it is more efficient to pass the components of the tuple as multiple arguments

or return values, the SML/NJ compiler flattens the arguments and the return values of a function. However, the flattening is done only if the corresponding tuples have less than 9 elements. The type system in the compiler isn't powerful enough to express these sort of transformations. The implementation gets around this in two ways. First, the function type constructor ($\twoheadrightarrow$) enjoys a "special" status: it includes a number of flags that indicate whether the argument or the result are candidates for flattening. These are term-level boolean flags, and are used for reifying the type translations to the meta-level. The implementation then checks the value of these flags and performs a number of *ad hoc* reductions (solely on the function type) to work around the problem. Second, a lot of the checking is done at the meta-level and not through the type system. For example, the type system does not enforce that only tuples with less than 9 elements will be flattened. Clearly, such an approach is unsatisfactory and we will show how our type system can obviate these disadvantages.

This example also shows how our type system generalizes previous approaches to intensional type analysis. In our system we can analyze tuples of arbitrary length, or function types with an arbitrary number of arguments or return values. This is possible because our mechanism for inductive definitions allows us to define a list of types $\Omega List$. We can now say that our tuple type constructor takes a list of types, and abstract away the length. All previous approaches to intensional type analysis were applicable only to restricted forms, such as pairs, or k-argument functions. In fact, Morrisett [Mor95] also shows how to use type analysis for flattening function arguments, but he considers functions with a fixed number of arguments.

We will consider only the type-level transformations; the term-level transformations are well known. Moreover, the SML/NJ compiler already implements the term-level flattening by using coercions in the manner of Shao [Sha97a]. Previous researchers have also used similar coercions for implementing optimizing compilers [SA95, Ler92].

We have already shown the inductive definition of natural numbers used in our implementation. We use the inductive mechanism to define a boolean kind as well, containing two nullary constructors true and false. We have also implemented a library of useful primitive recursive functions on values of these inductive kinds. Note that these are type-level functions operating on type-level values. For example, on natural numbers we provide functions for addition or comparison; on booleans we provide functions for doing conjunction, disjunction, or negation.

$$
\begin{array}{lll}
\mathsf{flatten} & = Fn(X{:}\Omega List, & \\
 & \quad Case\ X & \\
 & \quad\quad \mathsf{nil} & = \mathsf{nil} \\
 & \quad\quad \mathsf{cons}(Y_1, Y_2) & = Case\ Y_2 \\
 & \quad\quad\quad\quad \mathsf{nil} & = \mathsf{fbranches}\ Y_1 \\
 & \quad\quad\quad\quad \_\_ & = X) \\
\mathsf{fbranches} & = Fn(X'{:}\Omega, & \\
 & \quad Case\ X' & \\
 & \quad\quad \gg (N, Y') & = Case\ N \\
 & \quad\quad\quad\quad \mathsf{zero} & = [X'] \qquad \text{(* void type *)} \\
 & \quad\quad\quad\quad \mathsf{succ}(N') & = Case\ (\mathsf{Leq}\ N'\ \overline{8}) \\
 & \quad\quad\quad\quad\quad\quad \mathsf{true} & = Y' \\
 & \quad\quad\quad\quad\quad\quad \mathsf{false} & = [X'] \\
 & \quad\quad \_\_ & = [X'])
\end{array}
$$

$$
\begin{array}{ll}
\text{where} & [X'] = \mathsf{cons}(X', \mathsf{nil}) \\
 & \overline{8} = \underbrace{\mathsf{succ}(\ldots\mathsf{succ}}_{8}(\mathsf{zero}))
\end{array}
$$

We define a type-level flatten operator as shown above. We have taken some liberties with the PTS syntax to make the presentation clearer: we use named variables instead of DeBruijn indices, elide some typing annotations, and use pattern matching syntax for case statements. The Leq function is part of the library of functions that we have implemented. We won't show the code here since it is straightforward (specially because we use a unary encoding).

A function with type $A_1 \rightarrow A_2$ has the type flatten $A_1 \rightarrow$ flatten A_2 after flattening. The flatten operator first checks that the function has 1 argument/return-value. If this is true, it calls the fbranches function. This function checks if the argument is a tuple, and has a length between 1 and 9, before flattening it. The checking is possible since we store the length in the type of a tuple.

Benchmark	No of lines	Description
Mlyacc	6030	parser generator for ML
Vliw	3679	instruction scheduler for a VLIW compiler
CML	6152	concurrent ML implementation by Reppy [Rep91]
Simple	915	a spherical fluid dynamics program
lexgen	1171	lexical analyzer for ML

Figure 6.4: Benchmark programs

Benchmark	New		Old		Ratio
	User	GC	User	GC	(New/Old)
Lexgen	5.82	1.52	3.48	0.97	1.64
Simple	4.47	0.70	3.70	0.88	1.15
Vliw	35.14	6.13	15.24	5.49	1.99
ML-Yacc	30.61	5.07	14.01	2.57	2.15
CML	17.51	2.85	5.27	0.69	3.41

Figure 6.5: Compilation times (seconds)

6.5 Performance measurement

In this section we compare the performance of our implementation with that of Standard ML of New Jersey v110.34. This is a highly-tuned industrial strength compiler that is extensively used inside Bell Labs and many other institutions. We will be comparing only the compilation statistics since both the compilers generate identical code. To recap, our implementation uses a type system based on the calculus of inductive constructions, while the current implementation uses a type system based on the simply typed lambda calculus. Our implementation uses inductively defined kinds, dependent and polymorphic kinds and supports primitive recursion at the type level.

The set of type-preserving phases is the same in both the compilers. We instrumented the compilers to measure the amount of type information generated during the compilation. We used the existing timing mechanism in the Standard ML of New Jersey implementation; the compilation times were measured with the instrumentation turned off. The measurements were taken on a Pentium II, 450 MHz machine running Linux with 128MB memory. Figure 6.5 describes the benchmark programs used in the measurements. As can be seen all of these are substantial real-world ML applications.

Figures 6.5 through 6.7 present the measurements. We use New to denote our implementation, and Old to denote the Standard ML of New Jersey v110.34 compiler. Figure 6.5 shows a

Benchmark	New	Old	Ratio (New/Old)
Lexgen	329.71	96.98	3.40
Simple	227.07	66.63	3.40
Vliw	886.92	222.86	3.97
ML-Yacc	2930.24	577.86	5.07
CML	1867.34	395.90	4.71

Figure 6.6: Generated type information (in KBytes)

Benchmark	Code Size	Type Size	Ratio (Type/Code)
Lexgen	96.15	18.38	19%
Simple	80.15	12.76	16%
Vliw	262.56	22.99	8%
ML-Yacc	348.62	206.50	59%
CML	125.97	149.03	118%

Figure 6.7: Ratio of type size to code size (in KBytes)

comparison of the compilation times. As the figures show, the maximum overhead incurred by our implementation is about a factor of three. Correspondingly Figure 6.6 shows that the amount of type information generated during the compilation process increases by about a factor of four.

The code generated by a type-preserving compiler contains type annotations embedded inside it. Figure 6.7 compares the size of these type annotations to the size of the generated code. Both the type size and the code size were measured at the end of the type-preserving phases of the compiler (just before closure conversion). In a type-based certifying compiler, the types serve as the proof of safety. Therefore, this table gives a measure of the size of the proof (of type-safety) compared to the size of the code. In all cases, the type size is a fraction of the code size. The ratio increases in the case of CML. We assume it happens because CML is heavily functorized which causes the type information to grow dramatically.

While the figures indicate that the new type system does incur an appreciable overhead, a better tuned implementation can remove a significant amount of this inefficiency. Our main motivation was to examine how an expressive type system scales to handle real-world ML applications. We will elaborate on a number of design decisions that can be changed to make the implementation significantly faster.

- Our definition of Ω includes a definition of $\Omega List$, and most of our constructors take a list of types as an argument. By using a more sophisticated encoding of $\Omega List$, the amount

of type information being constructed can be reduced. Suppose for example that we want to build a sum type $\mathsf{int} + \mathsf{real}$. In our implementation we would then construct the term $+ (\mathsf{cons}(\mathsf{int}_{pts}, \mathsf{cons}(\mathsf{real}_{pts}, \mathsf{nil})))$. A list of n types requires 2n+1 nodes in this representation.

We could instead use the following representation for a list of types:

$$\begin{aligned}
\mathsf{Inductive}\ \Omega List : \mathsf{Kind} := \mathsf{nil} \quad & : \Omega List \\
| \ \mathsf{cons1} & : \Omega \to \Omega List \\
| \ \mathsf{cons2} & : \Omega \to \Omega \to \Omega List \\
| \ \mathsf{cons3} & : \Omega \to \Omega \to \Omega \to \Omega List \\
& \ldots \\
| \ \mathsf{cons} \quad & : \Omega \to \Omega List \to \Omega List
\end{aligned}$$

We have separate constructors till a particular value of n, say m. Lists of length smaller than m only require an extra node now. Experiments show that the list of types is usually small; thus only a few constructors would be sufficient.

- In the old compiler type constructors were built-in primitives; they were implemented as different branches of a datatype. The compiler used pattern matching to check the form of the constructor at the head: whether it was a function, tuple, sum, array, vector, *etc*. This form of pattern matching is specially pervasive in the code that deals with type-directed optimizations. In the new implementation the type constructors are members of an inductive definition. The compiler code can no longer use pattern matching to detect the form of a constructor. The compiler optimizations can be re-written to obviate this problem, but this would have involved a major redesign of almost the entire type-directed part of the compiler. Instead we provided some utility functions that simulate pattern matching. These functions are executed at every pattern matching site in the code; measurements show that this incurs a considerable overhead.

- For simplicity our type system used a unary encoding of the natural numbers. Again this introduces a considerable overhead, specially during arithmetic operations. We could remove this by using the binary encoding of the Coq [HPM+00] math library.

- The type system in the current compiler uses multiple argument functions. Our functions

on the other hand (the Fn construct) are single argument functions. Using multiple argument functions is much more efficient, specially in mutually recursive inductive definitions. In essence our implementation has to simulate multiple argument functions by passing the arguments as a record of types, and selecting from this record. Since Ω is a mutually recursive inductive definition, and is ubiquitous, this adds a considerable overhead to the type manipulations.

6.6 Related Work

Many of the ideas in our implementation have been adapted from Shao *et al.*'s excellent work [SLM98] on implementing typed intermediate languages. They were the first to demonstrate that typed intermediate languages can be used in industrial-strength compilers. Several other compilers such as TIL [TMC+96], GHC [Pey92], and ML-Kit [Tof98] have demonstrated the benefits of using typed intermediate languages. The suspension based lambda encoding used in our work is borrowed directly from the work of Nadathur [Nad94, NW90] on efficient lambda representations. He has also used his encoding in the λ-Prolog system. Shao and Appel [SA95] also used hash-consing to enforce dag-representation for types.

We adapted a lot of the theoretical underpinnings of our implementation from the Coq [HPM+00] system. The combination of the Fix and the $Case$ construct used for simulating primitive recursion is borrowed directly from there. The Coq system in turn uses the formalism by Gimenez [Gim98]. Our inductive definition mechanism is more sophisticated than the one presented in Chapter 5; in this case also we used the formalism of the Coq system.

Chapter 7

Conclusions and Future Work

Certifying compilation is a promising approach for implementing secure computing systems, since it provides a static and incontrovertible proof of a program's safety. Type-preserving compilation has emerged as the practical basis for constructing a certifying system leading to the concept of type-based certifying compilers. In a type-based certifying compiler, the type system for the intermediate language is used to encode safety properties; the type annotations in the generated code serve as the proof of safety.

Modern programming paradigms are increasingly giving rise to applications that require runtime type analysis. Services such as garbage collection, dynamic linking and reflection, marshalling, and type-safe persistent programming all analyze types to various degrees at runtime. Existing compilers use an untyped intermediate language for compiling code that involves runtime type inspection. They reify types into values and discard type information at some early stage during compilation. Unfortunately, this approach cannot be used in type-based certifying compilers; such compilers must support runtime type analysis in a type-safe manner.

Writing these type-analyzing applications in a certifying framework is desirable for a number of reasons. A computing system is only as secure as the services that it provides: typically these services include type-analyzing applications like garbage collection, linking, reflection, *etc*. Moreover, these applications often involve complex coding that can lead to subtle bugs. There is also a software engineering benefit. For example, in a type-safe garbage collector, the interface and invariants must be made explicit through the types. Type-checking now ensures that these invariants are respected whenever the collector is invoked.

This dissertation describes a type system that can be used for supporting runtime type analysis

in type-based certifying compilers. The type system has two novel features. First, it supports the analysis of quantified types. Second, it supports the explicit representation of logical proofs and propositions.

7.1 Summary of contributions

The core contribution of this dissertation is a type system for analyzing the type of runtime values, but this system has other important ramifications. The type system can be used for type-checking the *copy* function in a stop-and-copy garbage collector, and thus provides a significant basis for writing provably type-safe garbage collectors. The underlying ideas can also be used for integrating logical assertions in a type system, and enforcing more sophisticated invariants. To sum up:

- We show the design of a type system that supports the analysis of quantified types, both at the type level and at the term level. We prove that the resulting type system is strongly normalizing and confluent. We also show an encoding of the calculus in a type-erasure semantics. We prove that this encoding is sound by establishing a correspondence with reductions in an untyped language.

- We show that this type system can be applied for writing the *copy* function in a copying garbage collector in a type-safe manner. We show that type-checking this function relies crucially on the ability to analyze quantified types, like existential types. We prove that the language in which the *copy* function is written is sound. Our formalization does exclude some features of an industrial-strength collector, nevertheless it represents a significant step towards designing a type system that can be used for realistic garbage collectors.

- We show that the ideas (underlying the analysis of quantified types) can be extended to integrate runtime type analysis with the explicit representation of logical proofs and propositions. Again, we prove that the resulting type system is strongly normalizing and confluent, and the underlying logic is consistent.

- We show empirically that the type system can be used in an industrial-strength compiler. For this, we implemented the system in an experimental version of the SML/NJ compiler and compared its performance with version 110.34 of the compiler. On a set of large benchmarks, our measurements show that the new type system incurs a reasonable compilation overhead.

7.2 Future work

The work presented in this dissertation is a first step towards the goal of building a practical infrastructure for certified type analyzing applications. It opens up many possible avenues for future work.

In this dissertation we showed how a type system for runtime type analysis can be used in building a type-safe copying garbage collector. But we could use this for many other applications. Type-safe serialization is a promising candidate for translation into the framework proposed in Chapter 3. One of the key problems here is the handling of abstract types; we need some mechanism to break abstraction barriers. This is similar to the problem we encountered in writing the polymorphic equality function where we have to compare two objects of existential type (Section 3.3.1). Modelling reflection in Java seems another promising candidate since it provides a facility that is similar to intensional type analysis. However, the framework presented in Chapter 3 relies on structural equivalence of types, whereas Java uses name equivalence of types. Moreover, translating Java into a type-preserving framework would require the use of recursive types. Extending intensional type analysis to handle recursive types still remains an open problem. In separate work [STS00a, TSS00] we have shown different methods of analyzing recursive types, but unfortunately, these approaches turn out to be impractical. We believe that the encoding of recursive types proposed by Shao [Sha01] (and shown in Chapter 6) will lead to a satisfactory solution, but the details remain to be worked out.

In Chapter 4 we showed how the *copy* function in a simple copying garbage collector can be written in a provably type-safe manner. However, there are still quite a few issues that need to be resolved before constructing a practical type-safe garbage collector. In separate work [MSS01] we have shown how generational collectors may be modelled. The solution assumes that we can maintain a separate region for mutable data that gets scanned at each collection. This assumption may not be feasible for object-oriented languages where side-effects are more frequent. We will have to support either card-marking or remembered sets. Our presentation also does not deal with the problem of cyclic data structures. An interesting area of work would be to model Cheney style collectors.

The work in Chapter 5 opens up many possibilities in the area of type-based language design and compilation. Our implementation preserves types through the different optimizations and discards them before closure conversion. In separate work [SSTP01] we have shown how to propagate

types through CPS conversion and closure conversion. One possible area of work is to propagate these advanced types down to the machine code. Can we use the expressivity to capture more invariants: for example to enforce fine-grained access policies during memory load and store instructions, or to enforce calling conventions for native/foreign methods ? Appel and Felten [AF00a] have shown that the PCC framework can be used for enforcing a wide variety of security policies such as correct sequencing of API calls and correct locking protocols; they use Church's higher order logic for this. Since our framework is powerful enough to support reasoning in higher-order predicate logic, it would be interesting to model similar invariants in our system. Another possible area of research is to augment a source language with the capability of expressing logical assertions. For many non-trivial invariants, the compiler would need hints from the programmer to prove that they are satisfied. It would be impractical for the programmer to insert these hints in the object code: we should rather provide a facility to add these assertions in the source program itself.

From a practical standpoint, our implementation can be made more robust. The measurements in Chapter 6 show that the compilation time with the new type system increases by a factor of 3-4. It should be possible to reduce this overhead significantly. In Chapter 6 we listed a number of design decisions that can be changed to obtain a much better performance, at the cost of making the implementation more involved. Most importantly, we want to use the implementation to build realistic certified applications; for example, a type-safe garbage collector. This will give us a true measure of the overhead involved in writing certified applications.

Finally, we want to apply this technology to the compilation of mainstream languages such as Java. The object oriented paradigm, and specially Java, present very different challenges from those encountered in compiling ML-like languages: efficient object encodings in a typed framework, name-based class hierarchy, mutually recursive classes, class loaders, dynamic loading, and finalization to name a few. We believe that the real test of this technology lies in its successful application to the compilation of such languages.

The long term goal of our research is to build an industrial-strength certifying framework. Every piece of code should be shipped with a formal and verifiable proof of its safety. In addition, the host system should have a really skelatal trusted computing base. The TCB will include the verifier and some bootstrapping code, but all the other services should be verified independently and loaded as libraries. We believe this is feasible and is the holy grail of secure computing.

Appendix A

Formal Properties of λ_i^ω

In this chapter, we formalize the properties of λ_i^ω. We show that the type system is sound with respect to the operational semantics. We also show that all reductions in the type language are terminating and confluent.

A.1 Soundness of λ_i^ω

The operational semantics for λ_i^ω are in Figure A.3. The reduction rules are standard except for the typecase construct. The typecase chooses a branch depending on the head constructor of the type being analyzed and passes the corresponding subtypes as arguments. For example, while analyzing the polymorphic type $\forall\,[\kappa]\,\tau$, it chooses the $e_\forall$ branch and applies it to the kind κ and the type function τ. If the type being analyzed is not in normal form, the typecase reduces the type to its unique normal form.

We prove soundness of the system by using contextual semantics in Wright/Felleisen style [WF92]. The evaluation contexts E are shown in Figure A.1. The reduction rules for the redexes r are shown in Figure 3.7. We assume unique variable names and environments are sets of variables. The notation $\vdash e:\tau$ is used as a shorthand for $\varepsilon;\varepsilon;\varepsilon \vdash e:\tau$.

Lemma A.1.1 *If* $\varepsilon;\varepsilon \vdash \nu : \Omega$, *then* ν *is one of* int, $\nu_1 \to \nu_2$, $\forall\,[\kappa]\,\nu_1$, *or* $\forall^+\nu_1$.

Proof Since ν is well-formed in an empty environment, it does not contain any free type or kind variables. Therefore ν can not be a ν^0 since the head of a ν^0 is a type variable. The lemma now follows by inspecting the remaining possibilities for ν. □

$$
\begin{array}{lll}
(value) & v & ::= i \mid \lambda x{:}\tau.\, e \mid \mathsf{fix}\, x{:}\tau.\, v \mid \Lambda\alpha{:}\kappa.\, v \mid \Lambda^{+} j.\, v \\
(context) & E & ::= [\,] \mid E\, e \mid v\, E \mid E\,[\tau] \mid E\,[\kappa]^{+} \\
(redex) & r & ::= (\lambda x{:}\tau.\, e)\, v \mid (\Lambda\alpha{:}\kappa.\, v)\,[\tau] \mid (\Lambda^{+} j.\, e)\,[\kappa]^{+} \\
 & & \quad \mid (\mathsf{fix}\, x{:}\tau.\, v)\, v' \mid (\mathsf{fix}\, x{:}\tau.\, v)\,[\tau'] \\
 & & \quad \mid (\mathsf{fix}\, x{:}\tau.\, v)\,[\kappa]^{+} \\
 & & \quad \mid \mathsf{typecase}[\tau]\ \tau'\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^{+}}) \\
 & & \quad \mid \mathsf{typecase}[\tau]\ \mathsf{int}\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^{+}}) \\
 & & \quad \mid \mathsf{typecase}[\tau]\ \tau \rightarrow \tau'\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^{+}}) \\
 & & \quad \mid \mathsf{typecase}[\tau]\ \forall\,[\kappa]\,\tau\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^{+}}) \\
 & & \quad \mid \mathsf{typecase}[\tau]\ \forall^{+}\tau\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^{+}})
\end{array}
$$

Figure A.1: Term contexts

$$
\begin{array}{lll}
\nu^0 & ::= & \alpha \mid \nu^0\,\nu \mid \nu^0\,[\kappa] \\
 & & \quad \mid \mathsf{Typerec}[\kappa]\ \nu^0\ \mathsf{of}\ (\nu_{\mathsf{int}};\ \nu_{\rightarrow};\ \nu_{\forall};\ \nu_{\forall^{+}}) \\
\nu & ::= & \nu^0 \mid \mathsf{int} \mid \twoheadrightarrow \mid (\twoheadrightarrow)\,\nu \mid (\twoheadrightarrow)\,\nu\,\nu' \\
 & & \quad \mid \forall \mid \forall\,[\kappa] \mid \forall\,[\kappa]\,\nu \mid \forall^{+} \mid \forall^{+}\nu \\
 & & \quad \mid \lambda\alpha{:}\kappa.\, \nu, \text{ where } \forall \nu^0.\ \nu \neq \nu^0\,\alpha \text{ or } \alpha \in \mathit{ftv}(\nu^0) \\
 & & \quad \mid \Lambda j.\, \nu, \text{ where } \forall \nu^0.\ \nu \neq \nu^0\,[j] \text{ or } j \in \mathit{fkv}(\nu^0)
\end{array}
$$

Figure A.2: Normal forms in the λ_i^{ω} type language

Lemma A.1.2 (Decomposition of terms) *If $\vdash e{:}\tau$, then either e is a value or it can be decomposed into unique E and r such that $e = E\,[r]$.*

This is proved by induction over the derivation of $\vdash e{:}\tau$, using Lemma A.1.1 in the case of the typecase construct.

Corollary A.1.3 (Progress) *If $\vdash e{:}\tau$, then either e is a value or there exists an e' such that $e \mapsto e'$.*

Proof By Lemma A.1.2, we know that if $\vdash e{:}\tau$ and e is not a value, then there exist some E and redex e_1 such that $e = E\,[e_1]$. Since e_1 is a redex, there exists a contraction e_2 such that $e_1 \rightsquigarrow e_2$. Therefore $e \mapsto e'$ for $e' = E\,[e_2]$. □

Lemma A.1.4 *If $\vdash E\,[e]{:}\tau$, then there exists a τ' such that $\vdash e{:}\tau'$, and for all e' such that $\vdash e'{:}\tau'$ we have $\vdash E\,[e']{:}\tau$.*

$$(\lambda x\!:\!\tau.\, e)\, v \rightsquigarrow [v/x]e \qquad (\mathsf{fix}\, x\!:\!\tau.\, v)\, v' \rightsquigarrow ([\mathsf{fix}\, x\!:\!\tau.\, v/x]v)\, v'$$

$$(\Lambda\alpha\!:\!\kappa.\, v)[\tau] \rightsquigarrow [\tau/\alpha]v \qquad (\mathsf{fix}\, x\!:\!\tau.\, v)[\tau] \rightsquigarrow ([\mathsf{fix}\, x\!:\!\tau.\, v/x]v)[\tau]$$

$$(\Lambda^{+} j.\, v)[\kappa]^{+} \rightsquigarrow [\kappa/j]v \qquad (\mathsf{fix}\, x\!:\!\tau.\, v)[\kappa]^{+} \rightsquigarrow ([\mathsf{fix}\, x\!:\!\tau.\, v/x]v)[\kappa]^{+}$$

$$\frac{e \rightsquigarrow e'}{e\, e_1 \rightsquigarrow e'\, e_1} \qquad \frac{e \rightsquigarrow e'}{v\, e \rightsquigarrow v\, e'} \qquad \frac{e \rightsquigarrow e'}{e[\tau] \rightsquigarrow e'[\tau]} \qquad \frac{e \rightsquigarrow e'}{e[\kappa]^{+} \rightsquigarrow e'[\kappa]^{+}}$$

$$\mathsf{typecase}[\tau]\ \mathsf{int}\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}}) \rightsquigarrow e_{\mathsf{int}}$$

$$\mathsf{typecase}[\tau]\ (\tau_1 \rightarrow \tau_2)\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}}) \rightsquigarrow e_{\rightarrow}\, [\tau_1]\, [\tau_2]$$

$$\mathsf{typecase}[\tau]\ (\forall\, [\kappa]\, \tau)\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}}) \rightsquigarrow e_{\forall}\, [\kappa]^{+}\, [\tau]$$

$$\mathsf{typecase}[\tau]\ (\forall^{+}\tau)\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}}) \rightsquigarrow e_{\forall^{+}}\, [\tau]$$

$$\frac{\varepsilon;\varepsilon \vdash \tau' \rightsquigarrow^{*} \nu'\!:\!\Omega \qquad \nu' \text{ is a normal form}}{\begin{array}{c}\mathsf{typecase}[\tau]\ \tau'\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}}) \rightsquigarrow \\ \mathsf{typecase}[\tau]\ \nu'\ \mathsf{of}\ (e_{\mathsf{int}};\, e_{\rightarrow};\, e_{\forall};\, e_{\forall^{+}})\end{array}}$$

Figure A.3: Operational semantics of λ_i^{ω}

Proof The proof is by induction on the derivation of $\vdash E\,[e]\!:\!\tau$. The different forms of E are handled similarly; we will show only one case here.

- **case** $E = E_1\, e_1$: We have that $\vdash (E_1\,[e])\, e_1\!:\!\tau$. By the typing rules, this implies that $\vdash E_1\,[e]\!:\!\tau_1 \rightarrow \tau$, for some τ_1. By induction, there exists a τ' such that $\vdash e\!:\!\tau'$ and for all e' such that $\vdash e'\!:\!\tau'$, we have that $\vdash E_1\,[e']\!:\!\tau_1 \rightarrow \tau$. Therefore $\vdash (E_1\,[e'])\, e_1\!:\!\tau$. □

As usual, the proof of soundness depends on several substitution lemmas; these are shown below. The proofs are fairly straightforward and proceed by induction on the derivation of the judgments. The notion of substitution is extended to environments in the usual way.

Lemma A.1.5 *If* $\mathcal{E}, j \vdash \kappa$ *and* $\mathcal{E} \vdash \kappa'$*, then* $\mathcal{E} \vdash [\kappa'/j]\kappa$.

Lemma A.1.6 *If* $\mathcal{E}, j; \Delta \vdash \tau\ :\ \kappa$ *and* $\mathcal{E} \vdash \kappa'$*, then* $\mathcal{E}; \Delta[\kappa'/j] \vdash [\kappa'/j]\tau\ :\ [\kappa'/j]\kappa$.

Lemma A.1.7 *If* $\mathcal{E}, j; \Delta; \Gamma \vdash e\ :\ \tau$ *and* $\mathcal{E} \vdash \kappa$*, then* $\mathcal{E}; \Delta[\kappa/j]; \Gamma[\kappa/j] \vdash [\kappa/j]e\ :\ [\kappa/j]\tau$.

Lemma A.1.8 *If* $\mathcal{E}; \Delta, \alpha\!:\!\kappa' \vdash \tau\ :\ \kappa$ *and* $\mathcal{E}; \Delta \vdash \tau'\ :\ \kappa'$*, then* $\mathcal{E}; \Delta \vdash [\tau'/\alpha]\tau\ :\ \kappa$.

Lemma A.1.9 *If* $\mathcal{E}; \Delta, \alpha{:}\kappa; \Gamma \vdash e : \tau$ *and* $\mathcal{E}; \Delta \vdash \tau' : \kappa$*, then*
$\mathcal{E}; \Delta; \Gamma[\tau'/\alpha] \vdash [\tau'/\alpha]e : [\tau'/\alpha]\tau$.

Proof We prove this by induction on the structure of e. We demonstrate the proof here only for a few cases; the rest follow analogously.

- **case** $e = e_1\,[\tau_1]$: We have that $\mathcal{E}; \Delta \vdash \tau' : \kappa$. and also that $\mathcal{E}; \Delta, \alpha{:}\kappa; \Gamma \vdash e_1\,[\tau_1] : \tau$. By the typing rule for a type application we get that

  $\mathcal{E}; \Delta, \alpha{:}\kappa; \Gamma \vdash e_1 : \forall\beta{:}\kappa_1.\,\tau_2$ and
  $\mathcal{E}; \Delta, \alpha{:}\kappa \vdash \tau_1 : \kappa_1$ and
  $\tau = [\tau_1/\beta]\tau_2$
  By induction on e_1,

  $$\mathcal{E}; \Delta; \Gamma[\tau'/\alpha] \vdash [\tau'/\alpha]e_1 : \forall\beta{:}\kappa_1.\,[\tau'/\alpha]\tau_2$$

  By Lemma A.1.8, $\mathcal{E}; \Delta \vdash [\tau'/\alpha]\tau_1 : \kappa_1$. Therefore

  $$\mathcal{E}; \Delta; \Gamma[\tau'/\alpha] \vdash ([\tau'/\alpha]e_1)\,[[\tau'/\alpha]\tau_1] : [[\tau'/\alpha]\tau_1/\beta]([\tau'/\alpha]\tau_2)$$

  But this is equivalent to

  $$\mathcal{E}; \Delta; \Gamma[\tau'/\alpha] \vdash ([\tau'/\alpha]e_1)\,[[\tau'/\alpha]\tau_1] : [\tau'/\alpha]([\tau_1/\beta]\tau_2)$$

- **case** $e = e_1\,[\kappa_1]^+$: We have that $\mathcal{E}; \Delta, \alpha{:}\kappa; \Gamma \vdash e_1\,[\kappa_1]^+ : \tau$ and $\mathcal{E}; \Delta \vdash \tau' : \kappa$. By the typing rule for kind application,

  $\mathcal{E}; \Delta, \alpha{:}\kappa; \Gamma \vdash e_1 : \forall j.\,\tau_1$ and
  $\tau = [\kappa_1/j]\tau_1$ and
  $\mathcal{E} \vdash \kappa_1$
  By induction on e_1,

  $$\mathcal{E}; \Delta; \Gamma \vdash [\tau'/\alpha]e_1 : \forall j.\,[\tau'/\alpha]\tau_1$$

  Therefore

  $$\mathcal{E}; \Delta; \Gamma \vdash ([\tau'/\alpha]e_1)\,[\kappa_1]^+ : [\kappa_1/j]([\tau'/\alpha]\tau_1)$$

  Since j does not occur free in τ',

  $$[\kappa_1/j]([\tau'/\alpha]\tau_1) = [\tau'/\alpha]([\kappa_1/j]\tau_1)$$

- **case** $e = \mathsf{typecase}[\tau_0]\ \tau_1\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^+})$: From the typing rules, we have that $\mathcal{E}; \Delta \vdash \tau' : \kappa$ and $\mathcal{E}; \Delta, \alpha{:}\kappa; \Gamma \vdash \mathsf{typecase}[\tau_0]\ \tau_1\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^+}) : \tau_0\,\tau_1$. Using Lemma A.1.8 on the kind derivation of τ_0 and τ_1, and the inductive assumption on the typing rules for the subterms we get,

  $\mathcal{E}; \Delta \vdash [\tau'/\alpha]\tau_0 : \Omega \rightarrow \Omega$ and

  $\mathcal{E}; \Delta \vdash [\tau'/\alpha]\tau_1 : \Omega$ and

  $\mathcal{E}; \Delta; \Gamma[\tau'/\alpha] \vdash [\tau'/\alpha]e_{\mathsf{int}} : [\tau'/\alpha](\tau_0\ \mathsf{int})$ and

  $\mathcal{E}; \Delta; \Gamma[\tau'/\alpha] \vdash [\tau'/\alpha]e_{\rightarrow} : [\tau'/\alpha](\forall\alpha_1{:}\Omega.\,\forall\alpha_2{:}\Omega.\,\tau_0\,(\alpha_1 \rightarrow \alpha_2))$ and

  $\mathcal{E}; \Delta; \Gamma[\tau'/\alpha] \vdash [\tau'/\alpha]e_{\forall} : [\tau'/\alpha](\forall^{+}j.\,\forall\alpha{:}j \rightarrow \Omega.\,\tau_0\,(\mathbf{\forall}\,[j]\,\alpha))$ and

  $\mathcal{E}; \Delta; \Gamma[\tau'/\alpha] \vdash [\tau'/\alpha]e_{\forall^+} : [\tau'/\alpha](\forall\alpha{:}\forall j.\,\Omega.\,\tau_0\,(\mathbf{\forall}^{+}\alpha))$

  The above typing judgments are equivalent to

  $\mathcal{E}; \Delta \vdash [\tau'/\alpha]\tau_0 : \Omega \rightarrow \Omega$ and

  $\mathcal{E}; \Delta \vdash [\tau'/\alpha]\tau_1 : \Omega$ and

  $\mathcal{E}; \Delta; \Gamma[\tau'/\alpha] \vdash [\tau'/\alpha]e_{\mathsf{int}} : ([\tau'/\alpha]\tau_0)\ \mathsf{int}$ and

  $\mathcal{E}; \Delta; \Gamma[\tau'/\alpha] \vdash [\tau'/\alpha]e_{\rightarrow} : \forall\alpha_1{:}\Omega.\,\forall\alpha_2{:}\Omega.\,([\tau'/\alpha]\tau_0)\,(\alpha_1 \rightarrow \alpha_2)$ and

  $\mathcal{E}; \Delta; \Gamma[\tau'/\alpha] \vdash [\tau'/\alpha]e_{\forall} : \forall^{+}j.\,\forall\alpha{:}j \rightarrow \Omega.\,([\tau'/\alpha]\tau_0)\,(\mathbf{\forall}\,[j]\,\alpha)$ and

  $\mathcal{E}; \Delta; \Gamma[\tau'/\alpha] \vdash [\tau'/\alpha]e_{\forall^+} : \forall\alpha{:}\forall j.\,\Omega.\,([\tau'/\alpha]\tau_0)\,(\mathbf{\forall}^{+}\alpha)$

  from which the statement of the lemma follows directly. □

Lemma A.1.10 *If* $\mathcal{E}; \Delta; \Gamma, x{:}\tau' \vdash e : \tau$ *and* $\mathcal{E}; \Delta; \Gamma \vdash e' : \tau'$*, then* $\mathcal{E}; \Delta; \Gamma \vdash [e'/x]e : \tau$.

Proof Proved by induction over the structure of e. The different cases are proved similarly. We will show only two cases here.

- **case** $e = \Lambda\alpha{:}\kappa.\,v$: We have that

  $\mathcal{E}; \Delta; \Gamma, x{:}\tau' \vdash \Lambda\alpha{:}\kappa.\,v : \forall\alpha{:}\kappa.\,\tau$ and

  $\mathcal{E}; \Delta; \Gamma \vdash e' : \tau'$

  Since e can always be alpha-converted, we assume that α is not previously defined in Δ. This implies $\mathcal{E}; \Delta, \alpha{:}\kappa; \Gamma, x{:}\tau' \vdash v : \tau$. Since α is not free in e', we have $\mathcal{E}; \Delta, \alpha{:}\kappa; \Gamma \vdash e' : \tau'$. By induction, $\mathcal{E}; \Delta, \alpha{:}\kappa; \Gamma \vdash [e'/x]v : \tau$. Hence $\mathcal{E}; \Delta; \Gamma \vdash \Lambda\alpha{:}\kappa.\,[e'/x]v : \forall\alpha{:}\kappa.\,\tau$.

- **case** $e = \mathsf{typecase}[\tau_0]\ \tau_1\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^+})$: We have that

$\mathcal{E}; \Delta; \Gamma \vdash e' \,:\, \tau'$ and

$\mathcal{E}; \Delta; \Gamma, x{:}\tau' \vdash \mathsf{typecase}[\tau_0]\ \tau_1\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^+}) \,:\, \tau_0\,\tau_1$

By the typecase typing rule we get

$\mathcal{E}; \Delta \vdash \tau_0 \,:\, \Omega \rightarrow \Omega$ and

$\mathcal{E}; \Delta \vdash \tau_1 \,:\, \Omega$ and

$\mathcal{E}; \Delta; \Gamma, x{:}\tau' \vdash e_{\mathsf{int}} \,:\, \tau_0\,\mathsf{int}$ and

$\mathcal{E}; \Delta; \Gamma, x{:}\tau' \vdash e_{\rightarrow} \,:\, \forall\alpha_1{:}\Omega.\,\forall\alpha_2{:}\Omega.\,\tau_0\,(\alpha_1 \rightarrow \alpha_2)$ and

$\mathcal{E}; \Delta; \Gamma, x{:}\tau' \vdash e_{\forall} \,:\, \forall^+ j.\,\forall\alpha{:}j \rightarrow \Omega.\,\tau_0\,(\forall\,[j]\,\alpha)$ and

$\mathcal{E}; \Delta; \Gamma, x{:}\tau' \vdash e_{\forall^+} \,:\, \forall\alpha{:}\forall j.\,\Omega.\,\tau_0\,(\forall^+\alpha)$

Applying the inductive hypothesis to each of the subterms $e_{\mathsf{int}}, e_{\rightarrow}, e_{\forall}, e_{\forall^+}$ yields directly the claim. □

Definition A.1.11 *e evaluates to e' (written $e \mapsto e'$) if there exist E, e_1, and e_2 such that $e = E\,[e_1]$ and $e' = E\,[e_2]$ and $e_1 \rightsquigarrow e_2$.*

Theorem A.1.12 (Subject reduction) *If $\vdash e{:}\tau$ and $e \mapsto e'$, then $\vdash e'{:}\tau$.*

Proof By Lemma A.1.2, e can be decomposed into unique E and unique redex e_1 such that $e = E\,[e_1]$. By definition, $e' = E\,[e_2]$ and $e_1 \rightsquigarrow e_2$. By Lemma A.1.4, there exists a τ' such that $\vdash e_1{:}\tau'$. By the same lemma, all we need to prove is that $\vdash e_2{:}\tau'$ holds. This is proved by considering each possible redex in turn. We will show only two cases, the rest follow similarly.

- **case** $e_1 = (\mathsf{fix}\,x{:}\tau_1.\,v)\,v'$: Then $e_2 = ([\mathsf{fix}\,x{:}\tau_1.\,v/x]v)\,v'$. We have that $\vdash (\mathsf{fix}\,x{:}\tau_1.\,v)\,v'{:}\tau'$. By the typing rules for term application we get that for some τ_2,

  $\vdash \mathsf{fix}\,x{:}\tau_1.\,v{:}\tau_2 \rightarrow \tau'$ and

  $\vdash v'{:}\tau_2$

  By the typing rule for fix we get that,

  $\vdash \tau_1 = \tau_2 \rightarrow \tau'$ and

  $\varepsilon; \varepsilon; \varepsilon, x{:}\tau_2 \rightarrow \tau' \vdash v \,:\, \tau_2 \rightarrow \tau'$

  Using Lemma A.1.10 and the typing rule for application, we obtain the desired judgment

  $\vdash ([\mathsf{fix}\,x{:}\tau_1.\,v/x]v)\,v'{:}\tau'$

- **case** $e_1 = \mathsf{typecase}[\tau_0]\ \tau_1\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^+})$: If τ_1 is not in normal form, then the expression e_1 reduces to $e_2 = \mathsf{typecase}[\tau_0]\ \nu_1\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^+})$, where

$$
\begin{array}{lll}
(\mathit{kinds}) & \kappa ::= \Omega \mid \kappa \to \kappa' \mid j \mid \forall j.\,\kappa \\
\\
(\mathit{types}) & \tau ::= \mathsf{int} \mid \twoheadrightarrow \mid \forall \mid \forall^{+} \\
& \quad \mid \alpha \mid \Lambda j.\,\tau \mid \lambda\alpha{:}\kappa.\,\tau \mid \tau\,[\kappa] \mid \tau\,\tau' \\
& \quad \mid \mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^{+}})
\end{array}
$$

Figure A.4: The λ_i^ω type language

$$
\begin{array}{lll}
(\beta_1) & ::= & (\lambda\alpha{:}\kappa.\,\tau)\,\tau' \rightsquigarrow [\tau'/\alpha]\tau \\
(\beta_2) & ::= & (\Lambda j.\,\tau)\,[\kappa] \rightsquigarrow [\kappa/j]\tau \\
(\eta_1) & ::= & \lambda\alpha{:}\kappa.\,\tau\,\alpha \rightsquigarrow \tau \qquad \alpha \notin \mathit{ftv}(\tau) \\
(\eta_2) & ::= & \Lambda j.\,\tau\,[j] \rightsquigarrow \tau \qquad j \notin \mathit{fkv}(\tau) \\
(t_1) & ::= & \mathsf{Typerec}[\kappa]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^{+}}) \rightsquigarrow \tau_{\mathsf{int}} \\
(t_2) & ::= & \mathsf{Typerec}[\kappa]\ (\tau_1 \to \tau_2)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^{+}}) \rightsquigarrow \\
& & \quad \tau_{\to}\,\tau_1\,\tau_2 \\
& & \quad (\mathsf{Typerec}[\kappa]\ \tau_1\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^{+}})) \\
& & \quad (\mathsf{Typerec}[\kappa]\ \tau_2\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^{+}})) \\
\\
(t_3) & ::= & \mathsf{Typerec}[\kappa]\ (\forall\,[\kappa_1]\,\tau_1)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^{+}}) \rightsquigarrow \\
& & \quad \tau_{\forall}\,[\kappa_1]\,\tau_1 \\
& & \quad (\lambda\alpha{:}\kappa_1.\,\mathsf{Typerec}[\kappa]\ (\tau_1\,\alpha)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^{+}})) \\
\\
(t_4) & ::= & \mathsf{Typerec}[\kappa]\ (\forall^{+}\tau_1)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^{+}}) \rightsquigarrow \\
& & \quad \tau_{\forall^{+}}\,\tau_1 \\
& & \quad (\Lambda j.\,\mathsf{Typerec}[\kappa]\ (\tau_1\,[j])\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^{+}}))
\end{array}
$$

Figure A.5: Type reductions

$\varepsilon;\varepsilon \vdash \tau_1 \mapsto^* \nu_1 : \Omega$. The latter implies $\varepsilon;\varepsilon \vdash \tau_0\,\tau_1 = \tau_0\,\nu_1 : \Omega$, hence $\vdash e_2 : \tau'$ follows directly from $\vdash e_1 : \tau'$.

If τ_1 is in normal form ν_1, by the second premise of the typing rule for typecase and Lemma A.1.1 we have four cases for ν_1. In each case the contraction has the desired type $\tau_0\,\nu_1$, according to the corresponding premises of the typecase typing rule and the rules for type and kind applications. □

A.2 Strong normalization

The type language is shown in Figure A.4. The single step reduction relation ($\tau \rightsquigarrow \tau'$) is shown in Figure A.5.

Lemma A.2.1 *If* $\mathcal{E}; \Delta \vdash \tau \,:\, \kappa$ *and* $\tau \rightsquigarrow \tau'$*, then* $\mathcal{E}; \Delta \vdash \tau' \,:\, \kappa$.

Proof (Sketch) The proof follows from a case analysis of the reduction relation ($\rightsquigarrow$). □

Lemma A.2.2 *If* $\tau_1 \rightsquigarrow \tau_2$*, then* $[\tau/\alpha]\tau_1 \rightsquigarrow [\tau/\alpha]\tau_2$.

Proof The proof is by enumerating each possible reduction from τ_1 to τ_2.

case β_1: In this case, $\tau_1 = (\lambda\beta\!:\!\kappa.\,\tau')\,\tau''$ and $\tau_2 = [\tau''/\beta]\tau'$. This implies that

$$[\tau/\alpha]\tau_1 = (\lambda\beta\!:\!\kappa.\,[\tau/\alpha]\tau')\,[\tau/\alpha]\tau''$$

This beta reduces to

$$[[\tau/\alpha]\tau''/\beta]([\tau/\alpha]\tau')$$

Since β does not occur free in τ, this is equivalent to

$$[\tau/\alpha]([\tau''/\beta]\tau')$$

case β_2: In this case, $\tau_1 = (\Lambda j.\,\tau')\,[\kappa]$ and $\tau_2 = [\kappa/j]\tau'$. We get that

$$[\tau/\alpha]\tau_1 = (\Lambda j.\,[\tau/\alpha]\tau')\,[\kappa]$$

This beta reduces to

$$[\kappa/j][\tau/\alpha]\tau'$$

Since j is not free in τ, this is equivalent to

$$[\tau/\alpha]([\kappa/j]\tau')$$

case η_1: In this case, $\tau_1 = \lambda\beta\!:\!\kappa.\,\tau'\,\beta$ and $\tau_2 = \tau'$ and β does not occur free in τ'. We get that

$$[\tau/\alpha]\tau_1 = \lambda\beta\!:\!\kappa.\,([\tau/\alpha]\tau')\,\beta$$

Since this is a capture avoiding substitution, β still does not occur free in $[\tau/\alpha]\tau'$. Therefore this eta reduces to $[\tau/\alpha]\tau'$.

case η_2: In this case, $\tau_1 = \Lambda j.\, \tau'\, [j]$ and $\tau_2 = \tau'$ and j does not occur free in τ'. We get that

$$[\tau/\alpha]\tau_1 = \Lambda j.\, ([\tau/\alpha]\tau')\, [j]$$

Since this is a capture avoiding substitution, j still does not occur free in $[\tau/\alpha]\tau'$. Therefore, this eta reduces to $[\tau/\alpha]\tau'$.

case t_1: $\tau_1 = \mathsf{Typerec}[\kappa]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ and $\tau_2 = \tau_{\mathsf{int}}$. We get that

$$[\tau/\alpha]\tau_1 = \mathsf{Typerec}[\kappa]\ \mathsf{int}\ \mathsf{of}\ ([\tau/\alpha]\tau_{\mathsf{int}};\ [\tau/\alpha]\tau_{\rightarrow};\ [\tau/\alpha]\tau_{\forall};\ [\tau/\alpha]\tau_{\forall^+})$$

But this reduces by the t_1 reduction to $[\tau/\alpha]\tau_{\mathsf{int}}$.

case t_2: $\tau_1 = \mathsf{Typerec}[\kappa]\ (\tau' \rightarrow \tau'')\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ and

$$\tau_2 = \tau_{\rightarrow}\, \tau'\, \tau''\, (\mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+}))\, (\mathsf{Typerec}[\kappa]\ \tau''\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+}))$$

We get that

$$[\tau/\alpha]\tau_1 = \mathsf{Typerec}[\kappa]\ ([\tau/\alpha]\tau' \rightarrow [\tau/\alpha]\tau'')\ \mathsf{of}\ ([\tau/\alpha]\tau_{\mathsf{int}};\ [\tau/\alpha]\tau_{\rightarrow};\ [\tau/\alpha]\tau_{\forall};\ [\tau/\alpha]\tau_{\forall^+})$$

This reduces by t_2 to

$$\begin{aligned}
&[\tau/\alpha]\tau_{\rightarrow}\, ([\tau/\alpha]\tau')\, ([\tau/\alpha]\tau'') \\
&\ (\mathsf{Typerec}[\kappa]\ ([\tau/\alpha]\tau')\ \mathsf{of}\ ([\tau/\alpha]\tau_{\mathsf{int}};\ [\tau/\alpha]\tau_{\rightarrow};\ [\tau/\alpha]\tau_{\forall};\ [\tau/\alpha]\tau_{\forall^+})) \\
&\ (\mathsf{Typerec}[\kappa]\ ([\tau/\alpha]\tau'')\ \mathsf{of}\ ([\tau/\alpha]\tau_{\mathsf{int}};\ [\tau/\alpha]\tau_{\rightarrow};\ [\tau/\alpha]\tau_{\forall};\ [\tau/\alpha]\tau_{\forall^+}))
\end{aligned}$$

But this is syntactically equal to $[\tau/\alpha]\tau_2$.

case t_3: $\tau_1 = \mathsf{Typerec}[\kappa]\ (\forall\, [\kappa']\, \tau')\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ and

$$\tau_2 = \tau_{\forall}\, [\kappa']\, \tau'\, (\lambda\beta{:}\kappa'.\, \mathsf{Typerec}[\kappa]\ (\tau'\, \beta)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+}))$$

We get that

$$[\tau/\alpha]\tau_1 = \mathsf{Typerec}[\kappa]\ (\forall\, [\kappa']\, [\tau/\alpha]\tau')\ \mathsf{of}\ ([\tau/\alpha]\tau_{\mathsf{int}};\ [\tau/\alpha]\tau_{\rightarrow};\ [\tau/\alpha]\tau_{\forall};\ [\tau/\alpha]\tau_{\forall^+})$$

This reduces by t_3 to

$$[\tau/\alpha]\tau_\forall\,[\kappa']\,([\tau/\alpha]\tau')$$
$$(\lambda\beta{:}\kappa'.\,\mathsf{Typerec}[\kappa]\,(([\tau/\alpha]\tau')\,\beta)\,\mathsf{of}\,([\tau/\alpha]\tau_{\mathsf{int}};\,[\tau/\alpha]\tau_\rightarrow;\,[\tau/\alpha]\tau_\forall;\,[\tau/\alpha]\tau_{\forall^+}))$$

But this is syntactically equivalent to $[\tau/\alpha]\tau_2$.

case t_4: $\tau_1 = \mathsf{Typerec}[\kappa]\,(\forall^+\tau')\,\mathsf{of}\,(\tau_{\mathsf{int}};\,\tau_\rightarrow;\,\tau_\forall;\,\tau_{\forall^+})$ and

$$\tau_2 = \tau_{\forall^+}\,\tau'\,(\Lambda j.\,\mathsf{Typerec}[\kappa]\,(\tau'\,[j])\,\mathsf{of}\,(\tau_{\mathsf{int}};\,\tau_\rightarrow;\,\tau_\forall;\,\tau_{\forall^+}))$$

We get that

$$[\tau/\alpha]\tau_1 = \mathsf{Typerec}[\kappa]\,(\forall^+[\tau/\alpha]\tau')\,\mathsf{of}\,([\tau/\alpha]\tau_{\mathsf{int}};\,[\tau/\alpha]\tau_\rightarrow;\,[\tau/\alpha]\tau_\forall;\,[\tau/\alpha]\tau_{\forall^+})$$

This reduces by t_4 to

$$[\tau/\alpha]\tau_{\forall^+}\,([\tau/\alpha]\tau')$$
$$(\Lambda j.\,\mathsf{Typerec}[\kappa]\,(([\tau/\alpha]\tau')\,[j])\,\mathsf{of}\,([\tau/\alpha]\tau_{\mathsf{int}};\,[\tau/\alpha]\tau_\rightarrow;\,[\tau/\alpha]\tau_\forall;\,[\tau/\alpha]\tau_{\forall^+}))$$

But this is syntactically equal to $[\tau/\alpha]\tau_2$. □

Lemma A.2.3 *If* $\tau_1 \rightsquigarrow \tau_2$, *then* $[\kappa'/j']\tau_1 \rightsquigarrow [\kappa'/j']\tau_2$.

Proof This is proved by case analysis of the type reduction relation.

case β_1: In this case, $\tau_1 = (\lambda\beta{:}\kappa.\,\tau')\,\tau''$ and $\tau_2 = [\tau''/\beta]\tau'$. This implies that

$$[\kappa'/j']\tau_1 = (\lambda\beta{:}[\kappa'/j']\kappa.\,[\kappa'/j']\tau')\,[\kappa'/j']\tau''$$

This beta reduces to

$$[[\kappa'/j']\tau''/\beta]([\kappa'/j']\tau')$$

But this is equivalent to

$$[\kappa'/j']([\tau''/\beta]\tau')$$

case β_2: In this case, $\tau_1 = (\Lambda j.\, \tau')\,[\kappa]$ and $\tau_2 = [\kappa/j]\tau'$. We get that

$$[\kappa'/j']\tau_1 = (\Lambda j.\, [\kappa'/j']\tau')\,[[\kappa'/j']\kappa]$$

This beta reduces to

$$[[\kappa'/j']\kappa/j][\kappa'/j']\tau'$$

Since j is not free in κ', this is equivalent to

$$[\kappa'/j']([\kappa/j]\tau')$$

case η_1: In this case, $\tau_1 = \lambda\beta{:}\kappa.\, \tau'\,\beta$ and $\tau_2 = \tau'$ and β does not occur free in τ'. We get that

$$[\kappa'/j']\tau_1 = \lambda\beta{:}[\kappa'/j']\kappa.\, ([\kappa'/j']\tau')\,\beta$$

Again β does not occur free in $[\kappa'/j']\tau'$. Therefore this eta reduces to $[\kappa'/j']\tau'$.

case η_2: In this case, $\tau_1 = \Lambda j.\, \tau'\,[j]$ and $\tau_2 = \tau'$ and j does not occur free in τ'. We get that

$$[\kappa'/j']\tau_1 = \Lambda j.\, ([\kappa'/j']\tau')\,[j]$$

Since this is a capture avoiding substitution, j still does not occur free in $[\kappa'/j']\tau'$. Therefore, this eta reduces to $[\kappa'/j']\tau'$.

case t_1: $\tau_1 = \mathsf{Typerec}[\kappa]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ and $\tau_2 = \tau_{\mathsf{int}}$. We get that

$$[\kappa'/j']\tau_1 = \mathsf{Typerec}[[\kappa'/j']\kappa]\ \mathsf{int}\ \mathsf{of}\ ([\kappa'/j']\tau_{\mathsf{int}};\ [\kappa'/j']\tau_{\rightarrow};\ [\kappa'/j']\tau_{\forall};\ [\kappa'/j']\tau_{\forall^+})$$

But this reduces by the t_1 reduction to $[\kappa'/j']\tau_{\mathsf{int}}$.

case t_2: $\tau_1 = \mathsf{Typerec}[\kappa]\ (\tau' \rightarrow \tau'')\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ and

$$\tau_2 = \tau_{\rightarrow}\,\tau'\,\tau''\,(\mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+}))\,(\mathsf{Typerec}[\kappa]\ \tau''\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+}))$$

We get that

$$[\kappa'/j']\tau_1 = \mathsf{Typerec}[[\kappa'/j']\kappa]\ ([\kappa'/j']\tau' \rightarrow [\kappa'/j']\tau'')\ \mathsf{of}$$
$$([\kappa'/j']\tau_{\mathsf{int}};\ [\kappa'/j']\tau_{\rightarrow};\ [\kappa'/j']\tau_{\forall};\ [\kappa'/j']\tau_{\forall^+})$$

This reduces by t_2 to

$$
\begin{array}{l}
[\kappa'/j']\tau_{\rightarrow}\,([\kappa'/j']\tau')\,([\kappa'/j']\tau'') \\
\quad(\mathsf{Typerec}[[\kappa'/j']\kappa]\,([\kappa'/j']\tau')\ \mathsf{of}\ ([\kappa'/j']\tau_{\mathsf{int}};\,[\kappa'/j']\tau_{\rightarrow};\,[\kappa'/j']\tau_{\forall};\,[\kappa'/j']\tau_{\forall^+})) \\
\quad(\mathsf{Typerec}[[\kappa'/j']\kappa]\,([\kappa'/j']\tau'')\ \mathsf{of}\ ([\kappa'/j']\tau_{\mathsf{int}};\,[\kappa'/j']\tau_{\rightarrow};\,[\kappa'/j']\tau_{\forall};\,[\kappa'/j']\tau_{\forall^+}))
\end{array}
$$

But this is syntactically equal to $[\kappa'/j']\tau_2$.

case t_3: $\tau_1 = \mathsf{Typerec}[\kappa]\,(\forall\,[\kappa_1]\,\tau')\ \mathsf{of}\ (\tau_{\mathsf{int}};\,\tau_{\rightarrow};\,\tau_{\forall};\,\tau_{\forall^+})$ and

$$\tau_2 = \tau_{\forall}\,[\kappa_1]\,\tau'\,(\lambda\beta{:}\kappa_1.\,\mathsf{Typerec}[\kappa]\,(\tau'\,\beta)\ \mathsf{of}\ (\tau_{\mathsf{int}};\,\tau_{\rightarrow};\,\tau_{\forall};\,\tau_{\forall^+}))$$

We get that

$$
\begin{array}{l}
[\kappa'/j']\tau_1 = \\
\mathsf{Typerec}[[\kappa'/j']\kappa]\,(\forall\,[[\kappa'/j']\kappa_1]\,[\kappa'/j']\tau')\ \mathsf{of} \\
\quad([\kappa'/j']\tau_{\mathsf{int}};\,[\kappa'/j']\tau_{\rightarrow};\,[\kappa'/j']\tau_{\forall};\,[\kappa'/j']\tau_{\forall^+})
\end{array}
$$

This reduces by t_3 to

$$
\begin{array}{l}
[\kappa'/j']\tau_{\forall}\,[[\kappa'/j']\kappa_1]\,([\kappa'/j']\tau') \\
\quad(\lambda\beta{:}[\kappa'/j']\kappa_1.\,\mathsf{Typerec}[[\kappa'/j']\kappa]\,(([\kappa'/j']\tau')\,\beta)\ \mathsf{of} \\
\quad([\kappa'/j']\tau_{\mathsf{int}};\,[\kappa'/j']\tau_{\rightarrow};\,[\kappa'/j']\tau_{\forall};\,[\kappa'/j']\tau_{\forall^+}))
\end{array}
$$

But this is syntactically equivalent to $[\kappa'/j']\tau_2$.

case t_4: $\tau_1 = \mathsf{Typerec}[\kappa]\,(\forall^+\tau')\ \mathsf{of}\ (\tau_{\mathsf{int}};\,\tau_{\rightarrow};\,\tau_{\forall};\,\tau_{\forall^+})$ and

$$\tau_2 = \tau_{\forall^+}\,\tau'\,(\Lambda j.\,\mathsf{Typerec}[\kappa]\,(\tau'\,[j])\ \mathsf{of}\ (\tau_{\mathsf{int}};\,\tau_{\rightarrow};\,\tau_{\forall};\,\tau_{\forall^+}))$$

We get that

$$
\begin{array}{l}
[\kappa'/j']\tau_1 = \\
\mathsf{Typerec}[[\kappa'/j']\kappa]\,(\forall^+[\kappa'/j']\tau')\ \mathsf{of}\ ([\kappa'/j']\tau_{\mathsf{int}};\,[\kappa'/j']\tau_{\rightarrow};\,[\kappa'/j']\tau_{\forall};\,[\kappa'/j']\tau_{\forall^+})
\end{array}
$$

This reduces by t_4 to

$[\kappa'/j']\tau_{\forall^+}\,([\kappa'/j']\tau')$
$(\Lambda j.\,\mathsf{Typerec}[[\kappa'/j']\kappa]\;(([\kappa'/j']\tau')\;[j])\;\mathsf{of}\;([\kappa'/j']\tau_{\mathsf{int}};\;[\kappa'/j']\tau_{\rightarrow};\;[\kappa'/j']\tau_{\forall};\;[\kappa'/j']\tau_{\forall^+}))$

But this is syntactically equal to $[\kappa'/j']\tau_2$. □

Definition A.2.4 *A type τ is strongly normalizable if every reduction sequence from τ terminates into a normal form (with no redexes). We use $\nu(\tau)$ to denote the length of the largest reduction sequence from τ to a normal form.*

Definition A.2.5 *We define neutral types, n, as*

$n_0 ::= \Lambda j.\,\tau \mid \lambda\alpha\!:\!\kappa.\,\tau$
$n ::= \alpha \mid n_0\,\tau \mid n\,\tau \mid n_0\,[\kappa] \mid n\,[\kappa]$
$\quad \mid \mathsf{Typerec}[\kappa]\;\tau\;\mathsf{of}\;(\tau_{\mathsf{int}};\;\tau_{\rightarrow};\;\tau_{\forall};\;\tau_{\forall^+})$

Definition A.2.6 *A reducibility candidate (also referred to as a candidate) of kind κ is a set $\mathcal{C}$ of types of kind κ such that*

1. *if $\tau \in \mathcal{C}$, then τ is strongly normalizable.*
2. *if $\tau \in \mathcal{C}$ and $\tau \rightsquigarrow \tau'$, then $\tau' \in \mathcal{C}$.*
3. *if τ is neutral and if for all τ' such that $\tau \rightsquigarrow \tau'$, we have that $\tau' \in \mathcal{C}$, then $\tau \in \mathcal{C}$.*

This implies that the candidates are never empty since if α has kind κ, then α belongs to candidates of kind κ.

Definition A.2.7 *Let κ be an arbitrary kind. Let $\mathcal{C}_\kappa$ be a candidate of kind κ. Let $\mathcal{C}_{\Omega\rightarrow\Omega\rightarrow\kappa\rightarrow\kappa\rightarrow\kappa}$ be a candidate of kind $\Omega \rightarrow \Omega \rightarrow \kappa \rightarrow \kappa \rightarrow \kappa$. Let $\mathcal{C}_{\forall j.\,(j\rightarrow\Omega)\rightarrow(j\rightarrow\kappa)\rightarrow\kappa}$ be a candidate of kind $\forall j.\,(j \rightarrow \Omega) \rightarrow (j \rightarrow \kappa) \rightarrow \kappa$. Let $\mathcal{C}_{(\forall j.\,\Omega)\rightarrow(\forall j.\,\kappa)\rightarrow\kappa}$ be a candidate of kind $(\forall j.\,\Omega) \rightarrow (\forall j.\,\kappa) \rightarrow \kappa$. We then define the set R_Ω of types of kind Ω as*

$\tau \in R_\Omega$ *iff*
$\quad \forall\tau_{\mathsf{int}} \in \mathcal{C}_\kappa$
$\quad \forall\tau_{\rightarrow} \in \mathcal{C}_{\Omega\rightarrow\Omega\rightarrow\kappa\rightarrow\kappa\rightarrow\kappa}$,
$\quad \forall\tau_{\forall} \in \mathcal{C}_{\forall j.\,(j\rightarrow\Omega)\rightarrow(j\rightarrow\kappa)\rightarrow\kappa}$,
$\quad \forall\tau_{\forall^+} \in \mathcal{C}_{(\forall j.\,\Omega)\rightarrow(\forall j.\,\kappa)\rightarrow\kappa}$
$\quad\quad \Rightarrow \mathsf{Typerec}[\kappa]\;\tau\;\mathsf{of}\;(\tau_{\mathsf{int}};\;\tau_{\rightarrow};\;\tau_{\forall};\;\tau_{\forall^+}) \in \mathcal{C}_\kappa$

Lemma A.2.8 *R_Ω is a candidate of kind Ω.*

Proof Suppose $\tau \in R_\Omega$. Suppose that the types τ_{int}, $\tau_\rightarrow$, $\tau_\forall$, and $\tau_{\forall^+}$ belong to the candidates $\mathcal{C}_\kappa$, $\mathcal{C}_{\Omega\rightarrow\Omega\rightarrow\kappa\rightarrow\kappa\rightarrow\kappa}$, $\mathcal{C}_{\forall j.\,(j\rightarrow\Omega)\rightarrow(j\rightarrow\kappa)\rightarrow\kappa}$, $\mathcal{C}_{(\forall j.\,\Omega)\rightarrow(\forall j.\,\kappa)\rightarrow\kappa}$ respectively, where the candidates are of the appropriate kinds (see definition A.2.7).

1. Consider $\tau' = \mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+})$. By definition this belongs to $\mathcal{C}_\kappa$. By property 1 of definition A.2.6, τ' is strongly normalizable and therefore τ must be strongly normalizable.

2. Consider $\tau' = \mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+})$. Suppose $\tau \rightsquigarrow \tau_1$. Then $\tau' \rightsquigarrow \mathsf{Typerec}[\kappa]\ \tau_1\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+})$. Since $\tau' \in \mathcal{C}_\kappa$, $\mathsf{Typerec}[\kappa]\ \tau_1\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+})$ belongs to $\mathcal{C}_\kappa$ by property 2 of definition A.2.6. Therefore, by definition, τ_1 belongs to R_Ω.

3. Suppose τ is neutral and for all τ_1 such that $\tau \rightsquigarrow \tau_1$, $\tau_1 \in R_\Omega$. Consider $\tau' = \mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+})$. Since we know that τ_{int}, $\tau_\rightarrow$, $\tau_\forall$, and $\tau_{\forall^+}$ are strongly normalizable, we can induct over $len = \nu(\tau_{\mathsf{int}}) + \nu(\tau_\rightarrow) + \nu(\tau_\forall) + \nu(\tau_{\forall^+})$. We will prove that for all values of len, $\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+})$ always reduces to a type that belongs to $\mathcal{C}_\kappa$; given that τ_{int}, $\tau_\rightarrow$, $\tau_\forall$, and $\tau_{\forall^+}$ belong to $\mathcal{C}_\kappa$, $\mathcal{C}_{\Omega\rightarrow\Omega\rightarrow\kappa\rightarrow\kappa\rightarrow\kappa}$, $\mathcal{C}_{\forall j.\,(j\rightarrow\Omega)\rightarrow(j\rightarrow\kappa)\rightarrow\kappa}$, and $\mathcal{C}_{(\forall j.\,\Omega)\rightarrow(\forall j.\,\kappa)\rightarrow\kappa}$ respectively (see definition A.2.7).

   - $len = 0$ Then $\tau' \rightsquigarrow \mathsf{Typerec}[\kappa]\ \tau_1\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+})$ is the only possible reduction since τ is neutral. By the assumption on τ_1, this belongs to $\mathcal{C}_\kappa$.

   - $len = k + 1$ For the inductive case, assume that the hypothesis is true for $len = k$. That is, for $len = k$, $\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+})$ always reduces to a type that belongs to $\mathcal{C}_\kappa$; given that τ_{int}, $\tau_\rightarrow$, $\tau_\forall$, and $\tau_{\forall^+}$ belong to $\mathcal{C}_\kappa$, $\mathcal{C}_{\Omega\rightarrow\Omega\rightarrow\kappa\rightarrow\kappa\rightarrow\kappa}$, $\mathcal{C}_{\forall j.\,(j\rightarrow\Omega)\rightarrow(j\rightarrow\kappa)\rightarrow\kappa}$, and $\mathcal{C}_{(\forall j.\,\Omega)\rightarrow(\forall j.\,\kappa)\rightarrow\kappa}$ respectively. This implies that for $len = k$, $\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+})$ belongs to $\mathcal{C}_\kappa$ (by property 3 of definition A.2.6). For $len = k + 1$, consider $\tau' = \mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+})$. This can reduce to $\mathsf{Typerec}[\kappa]\ \tau_1\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+})$ which belongs to $\mathcal{C}_\kappa$. The other possible reductions are to $\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau'_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+})$ where $\tau_{\mathsf{int}} \rightsquigarrow \tau'_{\mathsf{int}}$, or to $\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau'_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+})$ where $\tau_\rightarrow \rightsquigarrow \tau'_\rightarrow$, or $\mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau'_\forall;\ \tau_{\forall^+})$ where $\tau_\forall \rightsquigarrow \tau'_\forall$, or

Typerec[κ] τ of $(\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau'_{\forall^+})$ where $\tau_{\forall^+} \rightsquigarrow \tau'_{\forall^+}$. By property 2 of definition A.2.6, each of τ'_{int}, $\tau'_{\rightarrow}$, $\tau'_{\forall}$, and $\tau'_{\forall^+}$ belongs to the required candidate and $len = k$ for each of the reducts. Therefore, by the inductive hypothesis, each of the reducts belongs to $\mathcal{C}_\kappa$.

Therefore Typerec[κ] τ of $(\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ always reduces to a type that belongs to $\mathcal{C}_\kappa$. By property 3 of definition A.2.6, Typerec[κ] τ of $(\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ also belongs to $\mathcal{C}_\kappa$. Therefore, $\tau \in R_\Omega$

□

Definition A.2.9 *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two candidates of kinds κ_1 and κ_2. We then define the set $\mathcal{C}_1 \rightarrow \mathcal{C}_2$, of types of kind $\kappa_1 \rightarrow \kappa_2$, as*
$\tau \in \mathcal{C}_1 \rightarrow \mathcal{C}_2$ *iff* $\forall \tau'(\tau' \in \mathcal{C}_1 \Rightarrow \tau\,\tau' \in \mathcal{C}_2)$

Lemma A.2.10 *If $\mathcal{C}_1$ and $\mathcal{C}_2$ are candidates of kinds κ_1 and κ_2, then $\mathcal{C}_1 \rightarrow \mathcal{C}_2$ is a candidate of kind $\kappa_1 \rightarrow \kappa_2$.*

Proof

1. Suppose τ of kind $\kappa_1 \rightarrow \kappa_2$ belongs to $\mathcal{C}_1 \rightarrow \mathcal{C}_2$. By definition, if $\tau' \in \mathcal{C}_1$, then $\tau\,\tau' \in \mathcal{C}_2$. Since $\mathcal{C}_2$ is a candidate, $\tau\,\tau'$ is strongly normalizable. Therefore, τ must be strongly normalizable since for every sequence of reductions $\tau \rightsquigarrow \tau_1 \ldots \tau_k \ldots$, there is a corresponding sequence of reductions $\tau\,\tau' \rightsquigarrow \tau_1\,\tau' \ldots \tau_k\,\tau' \ldots$.

2. Suppose τ of kind $\kappa_1 \rightarrow \kappa_2$ belongs to $\mathcal{C}_1 \rightarrow \mathcal{C}_2$ and $\tau \rightsquigarrow \tau'$. Suppose $\tau_1 \in \mathcal{C}_1$. By definition, $\tau\,\tau_1 \in \mathcal{C}_2$. But $\tau\,\tau_1 \rightsquigarrow \tau'\,\tau_1$. By using property 2 of definition A.2.6 on $\mathcal{C}_2$, $\tau'\,\tau_1 \in \mathcal{C}_2$; therefore, $\tau' \in \mathcal{C}_1 \rightarrow \mathcal{C}_2$.

3. Consider a neutral τ of kind $\kappa_1 \rightarrow \kappa_2$. Suppose that for all τ' such that $\tau \rightsquigarrow \tau'$, $\tau' \in \mathcal{C}_1 \rightarrow \mathcal{C}_2$. Consider $\tau\,\tau_1$ where $\tau_1 \in \mathcal{C}_1$. Since τ_1 is strongly normalizable, we can induct over $\nu(\tau_1)$. If $\nu(\tau_1) = 0$, then $\tau\,\tau_1 \rightsquigarrow \tau'\,\tau_1$. But $\tau'\,\tau_1 \in \mathcal{C}_2$ (by assumption on τ'), and since τ is neutral, no other reduction is possible. If $\nu(\tau_1) \neq 0$, then $\tau_1 \rightsquigarrow \tau'_1$. In this case, $\tau\,\tau_1$ may reduce to either $\tau'\,\tau_1$ or to $\tau\,\tau'_1$. We saw that the first reduct belongs to $\mathcal{C}_2$. By property 2 of definition A.2.6, $\tau'_1 \in \mathcal{C}_1$ and $\nu(\tau'_1) < \nu(\tau_1)$. By the inductive assumption

over $\nu(\tau_1)$, we get that $\tau\,\tau_1'$ belongs to $\mathcal{C}_2$. By property 3 of definition A.2.6, $\tau\,\tau_1 \in \mathcal{C}_2$. This implies that $\tau \in \mathcal{C}_1 \rightarrow \mathcal{C}_2$.

□

Definition A.2.11 *We use $\overline{j}$ to denote the set $j_1, \ldots, j_n$ of j. We use a similar syntax to denote a set of other constructs.*

Definition A.2.12 *Let $\kappa[\overline{j}]$ be a kind where $\overline{j}$ contains all the free kind variables of κ. Let $\overline{\kappa}$ be a sequence of closed kinds of the same length and $\overline{\mathcal{C}}$ be a sequence of candidates of the corresponding kind. We now define the set $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$ of types of kind $[\overline{\kappa}/\overline{j}]\kappa$ as*

1. *if $\kappa = \Omega$, then $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}] = R_\Omega$.*
2. *if $\kappa = j_i$, then $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}] = \mathcal{C}_i$.*
3. *if $\kappa = \kappa_1 \rightarrow \kappa_2$, then $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}] = \mathcal{S}_{\kappa_1}[\overline{\mathcal{C}}/\overline{j}] \rightarrow \mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{j}]$.*
4. *if $\kappa = \forall j.\, \kappa'$, then $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}] =$ the set of types τ of kind $[\overline{\kappa}/\overline{j}]\kappa$ such that for every kind κ'' and reducibility candidate $\mathcal{C}''$ of this kind, $\tau\,[\kappa''] \in \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}''/\overline{j}, j]$.*

Lemma A.2.13 *$\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$ is a reducibility candidate of kind $[\overline{\kappa}/\overline{j}]\kappa$.*

Proof For $\kappa = \Omega$, the lemma follows from lemma A.2.8. For $\kappa = j$, the lemma follows by definition. If $\kappa = \kappa_1 \rightarrow \kappa_2$, then the lemma follows from the inductive hypothesis on κ_1 and κ_2 and lemma A.2.10. We only need to prove the case for $\kappa = \forall j'.\, \kappa'$. We will induct over the size of κ with the $\overline{j}$ containing all the free kind variables of κ.

1. Consider a $\tau \in \mathcal{S}_{\forall j'.\, \kappa'}[\overline{\mathcal{C}}/\overline{j}]$. By definition, for any kind κ_1 and corresponding candidate $\mathcal{C}'$, $\tau\,[\kappa_1] \in \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$. Applying the inductive hypothesis on κ', we get that $\mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$ is a candidate. Therefore, $\tau\,[\kappa_1]$ is strongly normalizable which implies that τ is strongly normalizable.

2. Consider a $\tau \in \mathcal{S}_{\forall j'.\, \kappa'}[\overline{\mathcal{C}}/\overline{j}]$ and $\tau \rightsquigarrow \tau_1$. For any kind κ_1 and corresponding candidate $\mathcal{C}'$, by definition, $\tau\,[\kappa_1] \in \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$. But $\tau\,[\kappa_1] \rightsquigarrow \tau_1\,[\kappa_1]$. By the inductive hypothesis on κ', we get that $\mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$ is a candidate. By property 2 of definition A.2.6, $\tau_1\,[\kappa_1] \in \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$. Therefore, $\tau_1 \in \mathcal{S}_{\forall j'.\, \kappa'}[\overline{\mathcal{C}}/\overline{j}]$.

3. Consider a neutral τ so that for all τ_1, such that $\tau \rightsquigarrow \tau_1$, $\tau_1 \in \mathcal{S}_{\forall j'.\,\kappa'}[\overline{\mathcal{C}}/\overline{j}]$. Consider $\tau\,[\kappa_1]$ for an arbitrary kind κ_1 and corresponding candidate $\mathcal{C}'$. We have that $\tau\,[\kappa_1] \rightsquigarrow \tau_1\,[\kappa_1]$. This is the only possible reduction since τ is neutral. By the assumption on τ_1, $\tau_1\,[\kappa_1] \in \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$. By the inductive hypothesis on κ', we get that $\mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$ is a candidate. By property 3 of definition A.2.6, $\tau\,[\kappa_1] \in \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$. Therefore $\tau \in \mathcal{S}_{\forall j'.\,\kappa'}[\overline{\mathcal{C}}/\overline{j}]$.

□

Lemma A.2.14 $\mathcal{S}_{[\kappa'/j']\kappa}[\overline{\mathcal{C}}/\overline{j}] = \mathcal{S}_{\kappa}[\overline{\mathcal{C}}, \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}/\overline{j}]/\overline{j}, j']$

Proof The proof is by induction over the structure of κ. We will show only the case for polymorphic kinds, the others follow directly by induction. Suppose $\kappa = \forall j''.\,\kappa''$. Then the LHS is the set of types τ of kind $[\overline{\kappa}/\overline{j}](\forall j''.\,[\kappa'/j']\kappa'')$ such that for every kind κ''' and corresponding candidate $\mathcal{C}'''$, $\tau\,[\kappa''']$ belongs to $\mathcal{S}_{[\kappa'/j']\kappa''}[\overline{\mathcal{C}}, \mathcal{C}'''/\overline{j}, j'']$. Applying the inductive hypothesis to κ'', this is equal to $\mathcal{S}_{\kappa''}[\overline{\mathcal{C}}, \mathcal{C}''', \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}, \mathcal{C}'''/\overline{j}, j'']/\overline{j}, j'', j']$. But j'' does not occur free in κ' (variables in κ' can always be renamed). Therefore, $\tau\,[\kappa''']$ belongs to $\mathcal{S}_{\kappa''}[\overline{\mathcal{C}}, \mathcal{C}''', \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}/\overline{j}]/\overline{j}, j'', j']$. The RHS consists of types τ' of kind $[\overline{\kappa}, [\overline{\kappa}/\overline{j}]\kappa'/\overline{j}, j'](\forall j''.\,\kappa'')$ such that for every kind κ''' and corresponding candidate $\mathcal{C}'''$, $\tau'\,[\kappa''']$ belongs to $\mathcal{S}_{\kappa''}[\overline{\mathcal{C}}, \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}/\overline{j}], \mathcal{C}'''/\overline{j}, j', j'']$. Also, the kind of τ' is equivalent to $[\overline{\kappa}/\overline{j}](\forall j''.\,[\kappa'/j']\kappa'')$. □

Proposition A.2.15 *From lemma A.2.13, we know that $\mathcal{S}_{\kappa}[\overline{\mathcal{C}}/\overline{j}]$ is a candidate of kind $[\overline{\kappa}/\overline{j}]\kappa$, that $\mathcal{S}_{\Omega\to\Omega\to\kappa\to\kappa\to\kappa}[\overline{\mathcal{C}}/\overline{j}]$ is a candidate of kind $[\overline{\kappa}/\overline{j}](\Omega \to \Omega \to \kappa \to \kappa \to \kappa)$, that $\mathcal{S}_{\forall j.\,(j\to\Omega)\to(j\to\kappa)\to\kappa}[\overline{\mathcal{C}}/\overline{j}]$ is a candidate of kind $[\overline{\kappa}/\overline{j}](\forall j.\,(j \to \Omega) \to (j \to \kappa) \to \kappa)$, and $\mathcal{S}_{(\forall j.\,\Omega)\to(\forall j.\,\kappa)\to\kappa}[\overline{\mathcal{C}}/\overline{j}]$ is a candidate of kind $[\overline{\kappa}/\overline{j}]((\forall j.\,\Omega) \to (\forall j.\,\kappa) \to \kappa)$. In the rest of the section, we will assume that the types τ_{int}, $\tau_{\to}$, $\tau_{\forall}$, and $\tau_{\forall^+}$ belong to the above candidates respectively.*

Lemma A.2.16 $\mathsf{int} \in R_{\Omega} = \mathcal{S}_{\Omega}[\overline{\mathcal{C}}/\overline{j}]$

Proof Consider $\tau = \mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+})$. The lemma holds if $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+})$ belongs to $\mathcal{S}_{\kappa}[\overline{\mathcal{C}}/\overline{\kappa}]$ is true; given that $\tau_{\mathsf{int}} \in \mathcal{S}_{\kappa}[\overline{\mathcal{C}}/\overline{j}]$, and $\tau_{\to} \in \mathcal{S}_{\Omega\to\Omega\to\kappa\to\kappa\to\kappa}[\overline{\mathcal{C}}/\overline{j}]$, and $\tau_{\forall} \in \mathcal{S}_{\forall j.\,(j\to\Omega)\to(j\to\kappa)\to\kappa}[\overline{\mathcal{C}}/\overline{j}]$, and $\tau_{\forall^+} \in \mathcal{S}_{(\forall j.\,\Omega)\to(\forall j.\,\kappa)\to\kappa}[\overline{\mathcal{C}}/\overline{j}]$.

Since τ_{int}, $\tau_{\rightarrow}$, $\tau_{\forall}$, and $\tau_{\forall^+}$ are strongly normalizable, we will induct over $len = \nu(\tau_{\mathsf{int}}) + \nu(\tau_{\rightarrow}) + \nu(\tau_{\forall}) + \nu(\tau_{\forall^+})$. We will prove that for all values of len, $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ always reduces to a type that belongs to $\mathcal{C}_\kappa$; given that the branches belong to the candidates as in proposition A.2.15.

- $len = 0$ Then $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ can reduce only to τ_{int} which by assumption belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{\kappa}]$.

- $len = k + 1$ For the inductive case, assume that the hypothesis holds true for $len = k$. That is, for $len = k$, $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ always reduces to a type that belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$; given that τ_{int}, $\tau_{\rightarrow}$, $\tau_{\forall}$, and $\tau_{\forall^+}$ belong to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$, $\mathcal{S}_{\Omega\rightarrow\Omega\rightarrow\kappa\rightarrow\kappa\rightarrow\kappa}[\overline{\mathcal{C}}/\overline{j}]$, $\mathcal{S}_{\forall j.\,(j\rightarrow\Omega)\rightarrow(j\rightarrow\kappa)\rightarrow\kappa}[\overline{\mathcal{C}}/\overline{j}]$, and to $\mathcal{S}_{(\forall j.\,\Omega)\rightarrow(\forall j.\,\kappa)\rightarrow\kappa}[\overline{\mathcal{C}}/\overline{j}]$. This implies that for $len = k$, the type $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$ (by property 3 of definition A.2.6). For $len = k + 1$, τ can reduce to τ_{int} which belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$. The other possible reductions are to $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \mathsf{int}\ \mathsf{of}\ (\tau'_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ where $\tau_{\mathsf{int}} \rightsquigarrow \tau'_{\mathsf{int}}$, or to $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau'_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ where $\tau_{\rightarrow} \rightsquigarrow \tau'_{\rightarrow}$, or to $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau'_{\forall};\ \tau_{\forall^+})$ where we have $\tau_{\forall} \rightsquigarrow \tau'_{\forall}$, or otherwise to $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau'_{\forall^+})$ where $\tau_{\forall^+} \rightsquigarrow \tau'_{\forall^+}$. By property 2 of definition A.2.6, each of τ'_{int}, $\tau'_{\rightarrow}$, $\tau'_{\forall}$, $\tau'_{\forall^+}$ belongs to the same candidate. Moreover, $len = k$ for each of the reducts. Therefore, by the inductive hypothesis, each of the reducts belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$.

Therefore, $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ always reduces to a type that belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$. By property 3 of definition A.2.6, $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ also belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$. Therefore, $\mathsf{int} \in R_\Omega$. □

Lemma A.2.17 $\twoheadrightarrow \in R_\Omega \rightarrow R_\Omega \rightarrow R_\Omega = \mathcal{S}_{\Omega\rightarrow\Omega\rightarrow\Omega}[\overline{\mathcal{C}}/\overline{j}]$.

Proof $\twoheadrightarrow \in R_\Omega \rightarrow R_\Omega \rightarrow R_\Omega$ if for all $\tau_1 \in R_\Omega$, we get that $(\twoheadrightarrow)\tau_1 \in R_\Omega \rightarrow R_\Omega$. This is true if for all $\tau_2 \in R_\Omega$, we get that $(\twoheadrightarrow)\tau_1\,\tau_2 \in R_\Omega$. This is true if $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ (\twoheadrightarrow)\tau_1\,\tau_2\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$ is true with the conditions in proposition A.2.15. Since τ_1, τ_2, τ_{int}, $\tau_{\rightarrow}$, $\tau_{\forall}$, and $\tau_{\forall^+}$ are strongly normalizable, we will induct over $len = \nu(\tau_1) + \nu(\tau_2) + \nu(\tau_{\mathsf{int}}) + \nu(\tau_{\rightarrow}) + \nu(\tau_{\forall}) + \nu(\tau_{\forall^+})$. We will prove that for all values of len, the type $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ ((\twoheadrightarrow)\tau_1\tau_2)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ always reduces to a type that

belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$; given that $\tau_1 \in R_\Omega$, and $\tau_2 \in R_\Omega$, and $\tau_{\mathsf{int}} \in \mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$, and
$\tau_\rightarrow \in \mathcal{S}_{\Omega\rightarrow\Omega\rightarrow\kappa\rightarrow\kappa\rightarrow\kappa}[\overline{\mathcal{C}}/\overline{j}]$, and $\tau_\forall \in \mathcal{S}_{\forall j.\,(j\rightarrow\Omega)\rightarrow(j\rightarrow\kappa)\rightarrow\kappa}[\overline{\mathcal{C}}/\overline{j}]$, and
$\tau_{\forall^+} \in \mathcal{S}_{(\forall j.\,\Omega)\rightarrow(\forall j.\,\kappa)\rightarrow\kappa}[\overline{\mathcal{C}}/\overline{j}]$. Consider $\tau = \mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ ((\twoheadrightarrow)\tau_1\,\tau_2)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+})$.

- $len = 0$ The only reduction of τ is

$$\tau' = \tau_\rightarrow\,\tau_1\,\tau_2\,(\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \tau_1\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+}))$$
$$(\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \tau_2\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+}))$$

  Since both τ_1 and τ_2 belong to R_Ω, we get directly that
  $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \tau_1\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+})$ as well as
  $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \tau_2\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\rightarrow;\ \tau_\forall;\ \tau_{\forall^+})$ belong to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$. This implies that τ' also belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$.

- $len = k + 1$ The other possible reductions come from the reduction of one of the individual types τ_1, τ_2, τ_{int}, $\tau_\rightarrow$, $\tau_\forall$, and $\tau_{\forall^+}$. The proof in this case is similar to the proof of the corresponding case in lemma A.2.16.

Since τ is neutral, by property 3 of definition A.2.6, τ belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$. □

Lemma A.2.18 *If for all* $\tau_1 \in \mathcal{S}_{\kappa_1}[\overline{\mathcal{C}}/\overline{j}]$, $[\tau_1/\alpha]\tau \in \mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{j}]$, *then*
$\lambda\alpha\!:\![\overline{\kappa}/\overline{j}]\kappa_1.\,\tau \in \mathcal{S}_{\kappa_1\rightarrow\kappa_2}[\overline{\mathcal{C}}/\overline{j}]$.

Proof Consider the neutral type $\tau' = (\lambda\alpha\!:\![\overline{\kappa}/\overline{j}]\kappa_1.\,\tau)\,\tau_1$. We have that τ_1 is strongly normalizable and $[\alpha'/\alpha]\tau$ is strongly normalizable. Therefore, τ is also strongly normalizable. We will induct over $len = \nu(\tau) + \nu(\tau_1)$. We will prove that for all values of len, the type $(\lambda\alpha\!:\![\overline{\kappa}/\overline{j}]\kappa_1.\,\tau)\,\tau_1$ always reduces to a type that belongs to $\mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{j}]$; given that $\tau_1 \in \mathcal{S}_{\kappa_1}[\overline{\mathcal{C}}/\overline{j}]$ and $[\tau_1/\alpha]\tau \in \mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{j}]$.

- $len = 0$ There are two possible reductions. A beta reduction yields $[\tau_1/\alpha]\tau$ which by assumption belongs to $\mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{j}]$. If $\tau = \tau_0\,\alpha$ and α does not occur free in τ_0, then we have an eta reduction to $\tau_0\,\tau_1$. But in this case $[\tau_1/\alpha]\tau = \tau_0\,\tau_1$.

- $len = k + 1$ For the inductive case, assume that the hypothesis is true for $len = k$. There are two additional reductions. The type τ' can reduce to $(\lambda\alpha\!:\![\overline{\kappa}/\overline{j}]\kappa_1.\,\tau)\,\tau_1''$ where $\tau_1 \rightsquigarrow \tau_1''$. By property 2 of definition A.2.6, τ_1'' belongs to $\mathcal{S}_{\kappa_1}[\overline{\mathcal{C}}/\overline{j}]$. Therefore, $[\tau_1''/\alpha]\tau$

belongs to $\mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{j}]$. Moreover, $len = k$. By the inductive hypothesis, $(\lambda\alpha{:}\kappa_1.\,\tau)\,\tau_1''$ always reduces to a type that belongs to $\mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{j}]$. By property 3 of definition A.2.6, $(\lambda\alpha{:}\kappa_1.\,\tau)\,\tau_1''$ belongs to $\mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{j}]$.

The other reduction of τ' is to $(\lambda\alpha{:}[\overline{\kappa}/\overline{j}]\kappa_1.\,\tau'')\,\tau_1$ where $\tau \rightsquigarrow \tau''$. By lemma A.2.2, $[\tau_1/\alpha]\tau \rightsquigarrow [\tau_1/\alpha]\tau''$. By property 2 of definition A.2.6, $[\tau_1/\alpha]\tau'' \in \mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{j}]$. Moreover, $len = k$ for the type τ'. Therefore, by the inductive hypothesis, $(\lambda\alpha{:}[\overline{\kappa}/\overline{j}]\kappa_1.\,\tau'')\,\tau_1$ always reduces to a type that belongs to $\mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{j}]$. By property 3 of definition A.2.6, $(\lambda\alpha{:}[\overline{\kappa}/\overline{j}]\kappa_1.\,\tau'')\,\tau_1$ belongs to $\mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{j}]$.

Therefore, the neutral type τ' always reduces to a type that belongs to $\mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{j}]$. By property 3 of definition A.2.6, $\tau' \in \mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{j}]$. Therefore, $\lambda\alpha{:}[\overline{\kappa}/\overline{j}]\kappa_1.\,\tau$ belongs to $\mathcal{S}_{\kappa_1}[\overline{\mathcal{C}}/\overline{j}] \rightarrow \mathcal{S}_{\kappa_2}[\overline{\mathcal{C}}/\overline{j}]$. This implies that $\lambda\alpha{:}[\overline{\kappa}/\overline{j}]\kappa_1.\,\tau$ belongs to $\mathcal{S}_{\kappa_1\rightarrow\kappa_2}[\overline{\mathcal{C}}/\overline{j}]$. □

Lemma A.2.19 $\forall \in \mathcal{S}_{\forall j.\,(j\rightarrow\Omega)\rightarrow\Omega}[\overline{\mathcal{C}}/\overline{j}]$.

Proof This is true if for any kind $[\overline{\kappa}/\overline{j}]\kappa_1$, $\forall\,[[\overline{\kappa}/\overline{j}]\kappa_1] \in \mathcal{S}_{(j\rightarrow\Omega)\rightarrow\Omega}[\overline{\mathcal{C}}, \mathcal{C}_{\kappa_1}/\overline{j}, j]$. This implies that

$$\forall\,[[\overline{\kappa}/\overline{j}]\kappa_1] \in \mathcal{S}_{j\rightarrow\Omega}[\overline{\mathcal{C}}, \mathcal{C}_{\kappa_1}/\overline{j}, j] \rightarrow \mathcal{S}_{\Omega}[\overline{\mathcal{C}}, \mathcal{C}_{\kappa_1}/\overline{j}, j]$$

This is true if for all $\tau \in \mathcal{S}_{j\rightarrow\Omega}[\overline{\mathcal{C}}, \mathcal{C}_{\kappa_1}/\overline{j}, j]$, it is true that $\forall\,[[\overline{\kappa}/\overline{j}]\kappa_1]\,\tau \in \mathcal{S}_{\Omega}[\overline{\mathcal{C}}, \mathcal{C}_{\kappa_1}/\overline{j}, j]$. This in turn implies that $\forall\,[[\overline{\kappa}/\overline{j}]\kappa_1]\,\tau \in R_\Omega$. This is true if
$\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ (\forall\,[[\overline{\kappa}/\overline{j}]\kappa_1]\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$ is true with the conditions in proposition A.2.15. Since each of the types τ, τ_{int}, $\tau_{\rightarrow}$, $\tau_{\forall}$, and $\tau_{\forall^+}$ belongs to a candidate, they are strongly normalizable. We will induct over
$len = \nu(\tau) + \nu(\tau_{\mathsf{int}}) + \nu(\tau_{\rightarrow}) + \nu(\tau_{\forall}) + \nu(\tau_{\forall^+})$. We will prove that for all values of len, the type
$\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ (\forall\,[[\overline{\kappa}/\overline{j}]\kappa_1]\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ always reduces to a type that belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$; given that $\tau \in \mathcal{S}_{j\rightarrow\Omega}[\overline{\mathcal{C}}, \mathcal{C}_{\kappa_1}/\overline{j}, j]$, and $\tau_{\mathsf{int}} \in \mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$, and $\tau_{\rightarrow} \in \mathcal{S}_{\Omega\rightarrow\Omega\rightarrow\kappa\rightarrow\kappa\rightarrow\kappa}[\overline{\mathcal{C}}/\overline{j}]$, and $\tau_{\forall} \in \mathcal{S}_{\forall j.\,(j\rightarrow\Omega)\rightarrow(j\rightarrow\kappa)\rightarrow\kappa}[\overline{\mathcal{C}}/\overline{j}]$, and $\tau_{\forall^+} \in \mathcal{S}_{(\forall j.\,\Omega)\rightarrow(\forall j.\,\kappa)\rightarrow\kappa}[\overline{\mathcal{C}}/\overline{j}]$.
Consider $\tau' = \mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ (\forall\,[[\overline{\kappa}/\overline{j}]\kappa_1]\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$

- $len = 0$ Then the only possible reduction of τ' is

$$\tau_1' = \tau_{\forall}\,[[\overline{\kappa}/\overline{j}]\kappa_1]\,\tau\,(\lambda\alpha{:}[\overline{\kappa}/\overline{j}]\kappa_1.\,\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\,\tau\,\alpha\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+}))$$

Consider $\tau'' = \mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \tau\ \alpha\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$. For all $\tau_1 \in \mathcal{C}_{\kappa_1}$, the type $[\tau_1/\alpha]\tau''$ reduces to the type $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \tau\ \tau_1\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$. By assumption, τ belongs to $\mathcal{S}_j[\overline{\mathcal{C}}, \mathcal{C}_{\kappa_1}/\overline{j}, j] \rightarrow \mathcal{S}_\Omega[\overline{\mathcal{C}}, \mathcal{C}_{\kappa_1}/\overline{j}, j]$. Therefore, τ belongs to $\mathcal{C}_{\kappa_1} \rightarrow R_\Omega$ which implies $\tau\,\tau_1 \in R_\Omega$. Therefore $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \tau\ \tau_1\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$. Therefore, by lemma A.2.18, (replacing $\mathcal{S}_{\kappa_1}[\overline{\mathcal{C}}/\overline{j}]$ with $\mathcal{C}_{\kappa_1}$ in the lemma), $\lambda\alpha\!:\![\overline{\kappa}/\overline{j}]\kappa_1.\ \mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \tau\ \alpha\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ belongs to $\mathcal{C}_{\kappa_1} \rightarrow \mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$.

By assumption, $\tau_\forall$ belongs to $\mathcal{S}_{\forall j.\,(j\rightarrow\Omega)\rightarrow(j\rightarrow\kappa)\rightarrow\kappa}[\overline{\mathcal{C}}/\overline{j}]$.
Therefore, $\tau_\forall\,[[\overline{\kappa}/\overline{j}]\kappa_1]$ belongs to $\mathcal{S}_{(j\rightarrow\Omega)\rightarrow(j\rightarrow\kappa)\rightarrow\kappa}[\overline{\mathcal{C}}, \mathcal{C}_{\kappa_1}/\overline{j}, j]$. This implies that $\tau_\forall\,[[\overline{\kappa}/\overline{j}]\kappa_1]\,\tau$ belongs to $\mathcal{S}_{(j\rightarrow\kappa)\rightarrow\kappa}[\overline{\mathcal{C}}, \mathcal{C}_{\kappa_1}/\overline{j}, j]$.

Consider $\mathcal{C} = \mathcal{S}_{(j\rightarrow\kappa)\rightarrow\kappa}[\overline{\mathcal{C}}, \mathcal{C}_{\kappa_1}/\overline{j}, j]$. Then $\mathcal{C}$ is equal to $\mathcal{S}_{j\rightarrow\kappa}[\overline{\mathcal{C}}, \mathcal{C}_{\kappa_1}/\overline{j}, j] \rightarrow \mathcal{S}_\kappa[\overline{\mathcal{C}}, \mathcal{C}_{\kappa_1}/\overline{j}, j]$. This is equivalent to $(\mathcal{C}_{\kappa_1} \rightarrow \mathcal{S}_\kappa[\overline{\mathcal{C}}, \mathcal{C}_{\kappa_1}/\overline{j}, j]) \rightarrow \mathcal{S}_\kappa[\overline{\mathcal{C}}, \mathcal{C}_{\kappa_1}/\overline{j}, j]$. But j does not occur free in κ. So the above can be written as $(\mathcal{C}_{\kappa_1} \rightarrow \mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]) \rightarrow \mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$. This implies that τ_1' belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$.

- $len = k + 1$ The other possible reductions come from the reduction of one of the individual types τ, τ_{int}, $\tau_{\rightarrow}$, $\tau_{\forall}$, and $\tau_{\forall^+}$. The proof in this case is similar to the proof of the corresponding case in lemma A.2.16.

Since τ' is neutral, by property 3 of definition A.2.6, τ' belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$. □

Lemma A.2.20 *If for every kind κ' and reducibility candidate $\mathcal{C}'$ of this kind, $[\kappa'/j']\tau \in \mathcal{S}_\kappa[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$, then $\Lambda j'.\,\tau \in \mathcal{S}_{\forall j'.\,\kappa}[\overline{\mathcal{C}}/\overline{j}]$.*

Proof Consider the neutral type $\tau' = (\Lambda j'.\,\tau)\,[\kappa']$ for an arbitrary kind κ'. Since $[j''/j']\tau$ is strongly normalizable, τ is strongly normalizable. We will induct over $len = \nu(\tau)$. We will prove that for all values of len, the neutral type $(\Lambda j'.\,\tau)\,[\kappa']$ always reduces to a type that belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$; given that $[\kappa'/j']\tau \in \mathcal{S}_\kappa[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$.

- $len = 0$ There are two possible reductions. A beta reduction yields $[\kappa'/j']\tau$ which by assumption belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$. If $\tau = \tau_0\,[j']$ and j' does not occur free in τ_0, then we have an eta reduction to $\tau_0\,[\kappa']$. But in this case $[\kappa'/j']\tau = \tau_0\,[\kappa']$.

- $len = k + 1$ For the inductive case, assume that the hypothesis is true for $len = k$. There is one additional reduction, $(\Lambda j'.\,\tau)\,[\kappa'] \rightsquigarrow (\Lambda j'.\,\tau_1)\,[\kappa']$ where $\tau \rightsquigarrow \tau_1$. By lemma A.2.3,

we know that $[\kappa'/j']\tau \rightsquigarrow [\kappa'/j']\tau_1$. By property 2 of definition A.2.6, $[\kappa'/j']\tau_1 \in \mathcal{S}_\kappa[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$. Moreover, $len = k$ for this reduct. Therefore, by the inductive hypothesis, $(\Lambda j'.\, \tau_1)\,[\kappa']$ always reduces to a type that belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$. By property 3 of definition A.2.6, $(\Lambda j'.\, \tau_1)\,[\kappa']$ belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$.

Therefore, the neutral type τ' always reduces to a type that belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$. By property 3 of definition A.2.6, $\tau' \in \mathcal{S}_\kappa[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$. Therefore, $\Lambda j'.\, \tau$ belongs to $\mathcal{S}_{\forall j'.\,\kappa}[\overline{\mathcal{C}}/\overline{j}]$. □

Lemma A.2.21 *If $\tau \in \mathcal{S}_{\forall j.\,\kappa}[\overline{\mathcal{C}}/\overline{j}]$, then for every kind $[\overline{\kappa}/\overline{j}]\kappa'$ $\tau\,[[\overline{\kappa}/\overline{j}]\kappa'] \in \mathcal{S}_{[\kappa'/j]\kappa}[\overline{\mathcal{C}}/\overline{j}]$.*

Proof By definition, $\tau\,[[\overline{\kappa}/\overline{j}]\kappa']$ belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j]$, for every kind $[\overline{\kappa}/\overline{j}]\kappa'$ and reducibility candidate $\mathcal{C}'$ of this kind. Set $\mathcal{C}' = \mathcal{S}_{\kappa'}[\overline{\mathcal{C}}/\overline{j}]$. Applying lemma A.2.14 leads to the result. □

Lemma A.2.22 $\Vdash^+ \in \mathcal{S}_{(\forall j.\,\Omega)\to\Omega}[\overline{\mathcal{C}}/\overline{j}]$.

Proof This is true if for all $\tau \in \mathcal{S}_{\forall j.\,\Omega}[\overline{\mathcal{C}}/\overline{j}]$, we have $\Vdash^+\tau \in R_\Omega$. This is true if $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ (\Vdash^+\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\to;\ \tau_\forall;\ \tau_{\forall^+})$ belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$ with the conditions in proposition A.2.15. Since all the types are strongly normalizable, we will induct over $len = \nu(\tau) + \nu(\tau_{\mathsf{int}}) + \nu(\tau_\to) + \nu(\tau_\forall) + \nu(\tau_{\forall^+})$. We will prove that for all values of len, the type $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ (\Vdash^+\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\to;\ \tau_\forall;\ \tau_{\forall^+})$ always reduces to a type that belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$; given that $\tau \in \mathcal{S}_{\forall j.\,\Omega}[\overline{\mathcal{C}}/\overline{j}]$, $\tau_{\mathsf{int}} \in \mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$, $\tau_\to \in \mathcal{S}_{\Omega\to\Omega\to\kappa\to\kappa\to\kappa}[\overline{\mathcal{C}}/\overline{j}]$, $\tau_\forall \in \mathcal{S}_{\forall j.\,(j\to\Omega)\to(j\to\kappa)\to\kappa}[\overline{\mathcal{C}}/\overline{j}]$, and $\tau_{\forall^+} \in \mathcal{S}_{(\forall j.\,\Omega)\to(\forall j.\,\kappa)\to\kappa}[\overline{\mathcal{C}}/\overline{j}]$. Consider $\tau' = \mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ (\Vdash^+\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\to;\ \tau_\forall;\ \tau_{\forall^+})$

- $len = 0$ Then the only possible reduction of τ' is

$$\tau_{\forall^+}\,\tau\,(\Lambda j.\,\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ (\tau\,[j])\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\to;\ \tau_\forall;\ \tau_{\forall^+}))$$

  Consider $\tau'' = \mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ (\tau\,[j])\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\to;\ \tau_\forall;\ \tau_{\forall^+})$. For an arbitrary kind κ', $[\kappa'/j]\tau''$ is equal to $\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ \tau\,[\kappa']\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\to;\ \tau_\forall;\ \tau_{\forall^+})$. By the assumption on τ, we get that $\tau\,[\kappa'] \in R_\Omega$. Therefore, by definition, $[\kappa'/j]\tau'' \in \mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$. Since j does not occur free in κ, we can write this as $[\kappa'/j]\tau'' \in \mathcal{S}_\kappa[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j]$ for a candidate $\mathcal{C}'$ of kind κ'. By lemma A.2.20 $\Lambda j.\,\mathsf{Typerec}[[\overline{\kappa}/\overline{j}]\kappa]\ (\tau\,[j])\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\to;\ \tau_\forall;\ \tau_{\forall^+})$ belongs to $\mathcal{S}_{\forall j.\,\kappa}[\overline{\mathcal{C}}/\overline{j}]$. By the assumptions on $\tau_{\forall^+}$ and τ, $\tau_{\forall^+}\,\tau\,(\Lambda j.\,\mathsf{Typerec}[\kappa]\ (\tau\,[j])\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\to;\ \tau_\forall;\ \tau_{\forall^+}))$ belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$.

- $len = k + 1$ The other possible reductions come from the reduction of one of the individual types τ, τ_{int}, $\tau_{\rightarrow}$, $\tau_{\forall}$, and $\tau_{\forall^+}$. The proof in this case is similar to the proof of the corresponding case in lemma A.2.16.

Since τ' is neutral, by property 3 of definition A.2.6, τ' belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$. □

We now come to the main result of this section.

Theorem A.2.23 (Candidacy) *Let τ be a type of kind κ. Suppose all the free type variables of τ are in $\alpha_1 \ldots \alpha_n$ of kinds $\kappa_1 \ldots \kappa_n$ and all the free kind variables of κ, $\kappa_1 \ldots \kappa_n$ are among $j_1 \ldots j_m$. If $\mathcal{C}_1 \ldots \mathcal{C}_m$ are candidates of kinds $\kappa'_1 \ldots \kappa'_m$ and $\tau_1 \ldots \tau_n$ are types of kind $[\overline{\kappa'}/\overline{j}]\kappa_1 \ldots [\overline{\kappa'}/\overline{j}]\kappa_n$ which are in $\mathcal{S}_{\kappa_1}[\overline{\mathcal{C}}/\overline{j}] \ldots \mathcal{S}_{\kappa_n}[\overline{\mathcal{C}}/\overline{j}]$, then $[\overline{\kappa'}/\overline{j}]\tau[\overline{\tau}/\overline{\alpha}]$ belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$.*

Proof The proof is by induction over the structure of τ.

1. The cases of int, $\twoheadrightarrow$, $\forall$, $\forall^+$are covered by lemmas A.2.16 A.2.17 A.2.19 A.2.22.

2. Suppose $\tau = \alpha_i$ and $\kappa = \kappa_i$. Then $[\overline{\kappa'}/\overline{j}]\tau[\overline{\tau}/\overline{\alpha}] = \tau_i$. By assumption, this belongs to $\mathcal{S}_{\kappa_i}[\overline{\mathcal{C}}/\overline{j}]$.

3. Suppose $\tau = \tau'_1\ \tau'_2$. Then $\tau'_1 : \kappa' \rightarrow \kappa$ for some kind κ' and $\tau'_2 : \kappa'$. By the inductive hypothesis, $[\overline{\kappa'}/\overline{j}]\tau'_1[\overline{\tau}/\overline{\alpha}]$ belongs to $\mathcal{S}_{\kappa' \rightarrow \kappa}[\overline{\mathcal{C}}/\overline{j}]$ and $[\overline{\kappa'}/\overline{j}]\tau'_2[\overline{\tau}/\overline{\alpha}]$ belongs to $\mathcal{S}_{\kappa'}[\overline{\mathcal{C}}/\overline{j}]$. Therefore, $([\overline{\kappa'}/\overline{j}]\tau'_1[\overline{\tau}/\overline{\alpha}])\ ([\overline{\kappa'}/\overline{j}]\tau'_2[\overline{\tau}/\overline{\alpha}])$ belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$.

4. Suppose $\tau = \tau'\ [\kappa']$. Then $\tau' : \forall j_1.\ \kappa_1$ and $\kappa = [\kappa'/j_1]\kappa_1$. By the inductive hypothesis, $[\overline{\kappa'}/\overline{j}]\tau'[\overline{\tau}/\overline{\alpha}]$ belongs to $\mathcal{S}_{\forall j_1.\ \kappa_1}[\overline{\mathcal{C}}/\overline{j}]$. By lemma A.2.21 $[\overline{\kappa'}/\overline{j}]\tau'[\overline{\tau}/\overline{\alpha}]\ [[\overline{\kappa'}/\overline{j}]\kappa']$ belongs to $\mathcal{S}_{[\kappa'/j_1]\kappa_1}[\overline{\mathcal{C}}/\overline{j}]$ which is equivalent to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$.

5. Suppose $\tau = \mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$. Then $\tau' : \Omega$, and $\tau_{\mathsf{int}} : \kappa$, and $\tau_{\rightarrow} : \Omega \rightarrow \Omega \rightarrow \kappa \rightarrow \kappa \rightarrow \kappa$, and $\tau_{\forall} : \forall j.\ (j \rightarrow \Omega) \rightarrow (j \rightarrow \kappa) \rightarrow \kappa$, and $\tau_{\forall^+} : (\forall j.\ \Omega) \rightarrow (\forall j.\ \kappa) \rightarrow \kappa$. By the inductive hypothesis $[\overline{\kappa'}/\overline{j}]\tau'[\overline{\tau}/\overline{\alpha}]$ belongs to R_Ω, and $[\overline{\kappa'}/\overline{j}]\tau_{\mathsf{int}}[\overline{\tau}/\overline{\alpha}]$ belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$, and $[\overline{\kappa'}/\overline{j}]\tau_{\rightarrow}[\overline{\tau}/\overline{\alpha}]$ belongs to $\mathcal{S}_{\Omega \rightarrow \Omega \rightarrow \kappa \rightarrow \kappa \rightarrow \kappa}[\overline{\mathcal{C}}/\overline{j}]$, and $[\overline{\kappa'}/\overline{j}]\tau_{\forall}[\overline{\tau}/\overline{\alpha}]$ belongs to $\mathcal{S}_{\forall j.\ (j \rightarrow \Omega) \rightarrow (j \rightarrow \kappa) \rightarrow \kappa}[\overline{\mathcal{C}}/\overline{j}]$, and $[\overline{\kappa'}/\overline{j}]\tau_{\forall^+}[\overline{\tau}/\overline{\alpha}]$ belongs to $\mathcal{S}_{(\forall j.\ \Omega) \rightarrow (\forall j.\ \kappa) \rightarrow \kappa}[\overline{\mathcal{C}}/\overline{j}]$. By definition of R_Ω,

$$\mathsf{Typerec}[[\overline{\kappa'}/\overline{j}]\kappa]\ [\overline{\kappa'}/\overline{j}]\tau'[\overline{\tau}/\overline{\alpha}]\ \mathsf{of}$$
$$([\overline{\kappa'}/\overline{j}]\tau_{\mathsf{int}}[\overline{\tau}/\overline{\alpha}];\ [\overline{\kappa'}/\overline{j}]\tau_{\rightarrow}[\overline{\tau}/\overline{\alpha}];\ [\overline{\kappa'}/\overline{j}]\tau_{\forall}[\overline{\tau}/\overline{\alpha}];\ [\overline{\kappa'}/\overline{j}]\tau_{\forall^+}[\overline{\tau}/\overline{\alpha}])$$

$$
\begin{array}{lll}
(context) & C ::= [\,] \mid \Lambda j.\, C \mid C\,[\kappa] \mid \lambda\alpha{:}\kappa.\, C \mid C\,\tau \mid \tau\, C \\
& \quad \mid \mathsf{Typerec}[\kappa]\ C\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+}) \\
& \quad \mid \mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (C;\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+}) \\
& \quad \mid \mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ C;\ \tau_{\forall};\ \tau_{\forall^+}) \\
& \quad \mid \mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ C;\ \tau_{\forall^+}) \\
& \quad \mid \mathsf{Typerec}[\kappa]\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ C)
\end{array}
$$

Figure A.6: Type contexts

belongs to $\mathcal{S}_\kappa[\overline{\mathcal{C}}/\overline{j}]$.

6. Suppose $\tau = \lambda\alpha'{:}\kappa'.\,\tau_1$. Then $\tau_1 : \kappa''$ where the free type variables of τ_1 are in $\alpha_1, \ldots, \alpha_n, \alpha'$ and $\kappa = \kappa' \rightarrow \kappa''$. By the inductive hypothesis, $[\overline{\kappa'}/\overline{j}]\tau_1[\overline{\tau}, \tau'/\overline{\alpha}, \alpha']$ belongs to $\mathcal{S}_{\kappa''}[\overline{\mathcal{C}}/\overline{j}]$ where τ' is of kind $[\overline{\kappa'}/\overline{j}]\kappa'$ and belongs to $\mathcal{S}_{\kappa'}[\overline{\mathcal{C}}/\overline{j}]$. This implies that $[\tau'/\alpha'](([\overline{\kappa'}/\overline{j}]\tau_1[\overline{\tau}/\overline{\alpha}])$ (since α' occurs free only in τ_1) belongs to $\mathcal{S}_{\kappa''}[\overline{\mathcal{C}}/\overline{j}]$. By lemma A.2.18, $\lambda\alpha' : [\overline{\kappa'}/\overline{j}]\kappa'.\,([\overline{\kappa'}/\overline{j}]\tau_1[\overline{\tau}/\overline{\alpha}])$ belongs to $\mathcal{S}_{\kappa' \rightarrow \kappa''}[\overline{\mathcal{C}}/\overline{j}]$.

7. Suppose $\tau = \Lambda j'.\,\tau'$. Then $\tau' : \kappa''$ and $\kappa = \forall j'.\,\kappa''$. By the inductive hypothesis, $[\overline{\kappa'}, \kappa'/\overline{j}, j']\tau'[\overline{\tau}/\overline{\alpha}]$ belongs to $\mathcal{S}_{\kappa''}[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$ for an arbitrary kind κ' and candidate $\mathcal{C}'$ of kind κ'. Since j' occurs free only in τ', we get that $[\kappa'/j'](([\overline{\kappa'}/\overline{j}]\tau'[\overline{\tau}/\overline{\alpha}])$ belongs to $\mathcal{S}_{\kappa''}[\overline{\mathcal{C}}, \mathcal{C}'/\overline{j}, j']$. By lemma A.2.20, $\Lambda j'.\,([\overline{\kappa'}/\overline{j}]\tau'[\overline{\tau}/\overline{\alpha}])$ belongs to $\mathcal{S}_{\forall j'.\,\kappa''}[\overline{\mathcal{C}}/\overline{j}]$.

□

Suppose SN_i is the set of strongly normalizable types of kind κ_i.

Corollary A.2.24 *All types are strongly normalizable.*

Proof Follows from theorem A.2.23 by putting $\mathcal{C}_i = SN_i$ and $\tau_i = \alpha_i$. □

A.3 Confluence

To prove confluence of the reduction in the type language of λ_i^ω, we first define the compatible extension $\mapsto$ of the one-step reduction $\rightsquigarrow$. Let the set of type contexts (ranged over by C) be defined inductively as shown in Figure A.6. A context is thus a "type term" with a hole $[\,]$; the term $C\,[\tau]$ is defined as the type obtained by replacing the hole in C by τ.

Definition A.3.1 *$\tau_1 \mapsto \tau_2$ iff there exist types τ_1' and τ_2' and a type context C such that $\tau_1 = C[\tau_1']$, $\tau_2 = C[\tau_2']$, and $\tau_1' \rightsquigarrow \tau_2'$.*

Let as usual $\mapsto^*$ denote the reflexive and transitive closure of $\mapsto$.

Lemma A.3.2 *If $\tau \mapsto \tau'$, then $C[\tau] \mapsto C[\tau']$.*

Proof From compositionality of contexts, *i.e.* since for all contexts C_1 and C_2 and types τ, $C_1[C_2[\tau]] = C[\tau]$ for some context C, which is constructed inductively on the structure of C_1. □

Corollary A.3.3 *If $\tau \mapsto^* \tau'$, then $C[\tau] \mapsto^* C[\tau']$.*

The following lemmas are proved by induction on the structure of contexts.

Lemma A.3.4 *If $\tau_1 \mapsto \tau_2$, then $[\tau/\alpha]\tau_1 \mapsto [\tau/\alpha]\tau_2$.*

Proof sketch Follows from Lemma A.2.2. □

Lemma A.3.5 *If $\tau_1 \mapsto \tau_2$, then $[\kappa/j]\tau_1 \mapsto [\kappa/j]\tau_2$.*

Proof sketch Follows from Lemma A.2.3. □

Lemma A.3.6 *If $\mathcal{E}; \Delta \vdash C[\tau] : \kappa$, then there exist $\mathcal{E}'$, Δ', and κ' such that $\mathcal{E}'; \Delta' \vdash \tau : \kappa'$; furthermore, if $\mathcal{E}'; \Delta' \vdash \tau' : \kappa'$, then $\mathcal{E}; \Delta \vdash C[\tau'] : \kappa$.*

By induction on the structure of types we prove the following substitution lemmas.

Lemma A.3.7 *If $\mathcal{E}; \Delta, \alpha{:}\kappa' \vdash \tau : \kappa$ and $\mathcal{E}; \Delta \vdash \tau' : \kappa'$, then $\mathcal{E}; \Delta \vdash [\tau'/\alpha]\tau : \kappa$.*

Lemma A.3.8 *If $\mathcal{E}, j; \Delta \vdash \tau : \kappa$ and $\mathcal{E} \vdash \kappa'$, then $\mathcal{E}; \Delta[\kappa'/j] \vdash [\kappa'/j]\tau : [\kappa'/j]\kappa$.*

Now we can show subject reduction for $\rightsquigarrow$.

Lemma A.3.9 *If $\mathcal{E}; \Delta \vdash \tau : \kappa$ and $\tau \rightsquigarrow \tau'$, then $\mathcal{E}; \Delta \vdash \tau' : \kappa$.*

Proof sketch Follows by case analysis of the reduction relation $\rightsquigarrow$ and the substitution Lemmas A.3.7 and A.3.8. □

Then we have subject reduction for $\mapsto$ as a corollary of Lemmas A.3.9 and A.3.6.

Corollary A.3.10 *If $\mathcal{E}; \Delta \vdash \tau : \kappa$ and $\tau \mapsto \tau'$, then $\mathcal{E}; \Delta \vdash \tau' : \kappa$.*

For our confluence proof we need another property of substitution.

Lemma A.3.11 *If* $\tau_1 \mapsto \tau_2$, *then* $[\tau_1/\alpha]\tau \mapsto^* [\tau_2/\alpha]\tau$.

Proof The proof is by induction on the structure of τ. The cases when τ is a constant, $\tau = \alpha$, or $\tau = \beta \neq \alpha$, are straightforward.

case $\tau = \Lambda j.\, \tau'$: Without loss of generality assume that j is not free in τ_1, so that $[\tau_1/\alpha]\tau = \Lambda j.\, ([\tau_1/\alpha]\tau')$; then by subject reduction (Corollary A.3.10) j is not free in τ_2, hence $[\tau_2/\alpha]\tau = \Lambda j.\, ([\tau_2/\alpha]\tau')$. By the induction hypothesis we have that $[\tau_1/\alpha]\tau' \mapsto^* [\tau_2/\alpha]\tau'$. Then by Corollary A.3.3 for the context $\Lambda j.\, [\,]$ we obtain $\Lambda j.\, ([\tau_1/\alpha]\tau') \mapsto^* \Lambda j.\, ([\tau_2/\alpha]\tau')$.

The cases of $\tau = \tau'\,[\kappa]$ and $\tau = \lambda\beta{:}\kappa.\, \tau'$ are similar.

case $\tau = \tau'\,\tau''$: By induction hypothesis we have

$$\begin{array}{lrcl} (1) & [\tau_1/\alpha]\tau' & \mapsto^* & [\tau_2/\alpha]\tau' \\ (2) & [\tau_1/\alpha]\tau'' & \mapsto^* & [\tau_2/\alpha]\tau''. \end{array}$$

Using context $[\,]\,([\tau_1/\alpha]\tau'')$, from (1) and Corollary A.3.3 it follows that

$$[\tau_1/\alpha]\tau = ([\tau_1/\alpha]\tau')\,([\tau_1/\alpha]\tau'') \mapsto^* ([\tau_2/\alpha]\tau')\,([\tau_1/\alpha]\tau'');$$

then using context $([\tau_2/\alpha]\tau')\,[\,]$, from (2) and Corollary A.3.3 we have

$$([\tau_2/\alpha]\tau')\,([\tau_1/\alpha]\tau'') \mapsto^* ([\tau_2/\alpha]\tau')\,(([\tau_2/\alpha]\tau'')) = [\tau_2/\alpha]\tau$$

and the result follows since $\mapsto^*$ is closed under transitivity.

The case of $\tau = \mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$ is similar. □

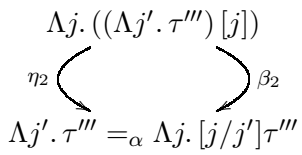
The next step is to prove local confluence of the reduction of well-formed types.

Lemma A.3.12 *If* $\mathcal{E}; \Delta \vdash \tau : \kappa_0$, $\tau \mapsto \tau_1$, *and* $\tau \mapsto \tau_2$, *then there exists* τ_0 *such that* $\tau_1 \mapsto^* \tau_0$ *and* $\tau_2 \mapsto^* \tau_0$.

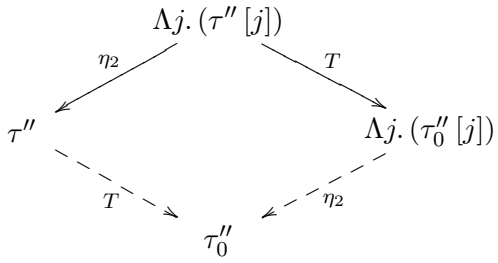
Proof The proof proceeds by induction on the structure of the derivation of $\mathcal{E}; \Delta \vdash \tau : \kappa_0$. For the base cases, corresponding to τ being one of the Ω constructors or a type variable, no rules of reduction apply, so the result is trivial. For the other cases, let $C_1, C_2, \tau_1', \tau_2', \tau_1''$, and τ_2'' be such that $\tau = C_1\,[\tau_1'] = C_2\,[\tau_2']$, $\tau_1 = C_1\,[\tau_1'']$, $\tau_2 = C_2\,[\tau_2'']$, and $\tau_1' \rightsquigarrow \tau_1''$, $\tau_2' \rightsquigarrow \tau_2''$.

case $\tau = \Lambda j.\, \tau'$: An inspection of the definition of contexts shows that the only possible forms for C_1 and C_2 are $[\,]$ and $\Lambda j.\, C$. Thus, accounting for the symmetry, there are the following three subcases:

- Both C_1 and C_2 are $[\,]$. The only reduction rule that applies then is η_2, so $\tau_1 = \tau_2$.
- $C_1 = \Lambda j.\, C_1'$ and $C_2 = \Lambda j.\, C_2'$. Then the result follows by the inductive hypothesis and Corollary A.3.3.
- $C_1 = [\,]$ and $C_2 = \Lambda j.\, C_2'$. Again. the only reduction for τ_1' is η_2, so $\tau' = \tau''\,[j]$ for some τ''. Then there are two cases for τ_2'. First, if $C_2' = [\,]$, then $\tau_2' = \tau'$, and—by inspection of the rules—in the case of kind application the only possible reduction is via β_2, hence $\tau'' = \Lambda j'.\, \tau'''$ for some j' and τ'''. Representing the reductions diagrammatically, we have immediate confluence (up to renaming of bound variables):

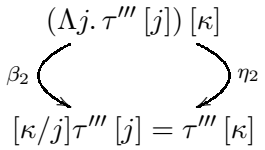
  

  The second case accounts for all other possibilities for C_2' (which must be of the form $C_2''\,[j]$) and reduction rules that can be applied in $\tau'' = C_2''\,[\tau_2']$ to reduce it (by assumption) to $C_2''\,[\tau_2'']$, which we denote by τ_0''. The dashed arrows show the reductions that complete local confluence.

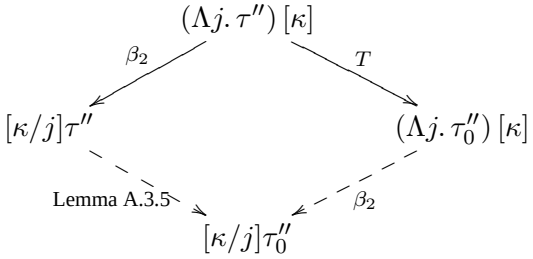
  

case $\tau = \tau'\,[\kappa]$: Again by inspection of the rules we have that the contexts are either empty or of the form $C\,[\kappa]$. The symmetric cases are handled as in the case of kind abstraction above. The interesting situation is when $C_1 = [\,]$ and $C_2 = C_2'\,[\kappa]$. The only reduction rule that applies for τ_1 is then β_2, hence $\tau' = \Lambda j.\, \tau''$ for some j and τ''. Again we have two major cases for τ_2: first, if

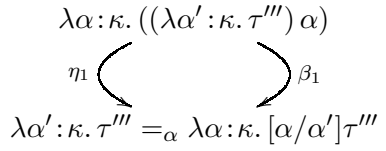
$C_2' = [\,]$, only η_2 applies, so $\tau'' = \tau'''\,[j]$ for some τ''', thus



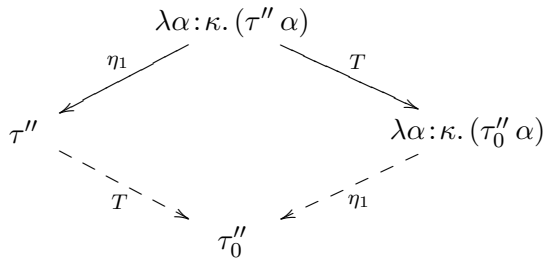
In all other cases we have $C_2' = \Lambda j.\,C_2''$, so $\tau'' = C_2''\,[\tau_2'] \mapsto C_2''\,[\tau_2'']$; letting τ_0'' stand for the latter, we have the diagram



case $\tau = \lambda\alpha\!:\!\kappa.\,\tau'$: The contexts can be either empty or of the form $\lambda\alpha\!:\!\kappa.\,C$. The symmetric cases are similar to those above. In the case when $C_1 = [\,]$ and $C_2 = \lambda\alpha\!:\!\kappa.\,C_2'$, the only rule that applies for the reduction of τ_1' is η_1, so $\tau' = \tau''\,\alpha$ for some τ''. Again, there are two cases for τ_2': First, if $C_2' = [\,]$, we have $\tau_2' = \tau' = \tau''\,\alpha$, and the only reduction rule for application is β_1, hence $\tau'' = \lambda\alpha'\!:\!\kappa'.\,\tau'''$ for some α', κ', and τ'''. Since $\mathcal{E}; \Delta \vdash \tau\ :\ \kappa_0$, the subterm $(\lambda\alpha'\!:\!\kappa'.\,\tau''')\,\alpha$ must be well-typed in an environment assigning kind κ to α, hence $\kappa' = \kappa$, so that



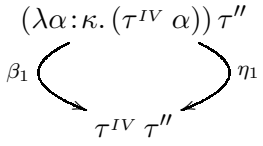
In all other cases for C_2' (which are of the form $C_2''\,\alpha$), we have $\tau'' = C_2''\,[\tau_2'] \mapsto C_2''\,[\tau_2'']$; denoting the latter type by τ_0'', we obtain



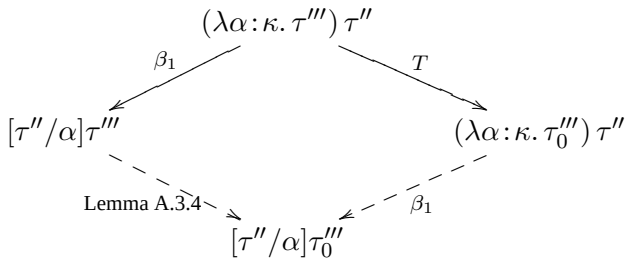
case $\tau = \tau'\,\tau''$: There are three possibilities for the contexts C_1 and C_2: to be empty, of the form $C\,\tau'$, or of the form $\tau\,C$. The symmetric cases proceed as before.

When $C_1 = C_1'\,\tau''$ and $C_2 = \tau'\,C_2'$, the redexes in τ_1' and τ_2' are in different subterms of the type, hence the reductions commute: we have $C_1'\,[\tau_1'] = \tau'$ and $C_2'\,[\tau_2'] = \tau''$, therefore $\tau_1 = (C_1'\,[\tau_1''])\,(C_2'\,[\tau_2'])$ and $\tau_2 = (C_1'\,[\tau_1'])\,(C_2'\,[\tau_2''])$, which both reduce to $(C_1'\,[\tau_1''])\,(C_2'\,[\tau_2''])$.

When $C_1 = [\,]$ and $C_2 = C_2'\,\tau''$, the only reduction rule that applies for $\tau_1' = \tau'\,\tau''$ is β_1, hence $\tau' = \lambda\alpha{:}\kappa.\,\tau'''$ for some α, κ, and τ'''. As before, there are two cases for C_2'. If it is empty, then the only reduction rule that applies to $\tau_2' = \tau'$ is η_1, hence $\tau''' = \tau^{IV}\,\alpha$ for some τ^{IV}, and local confluence follows by

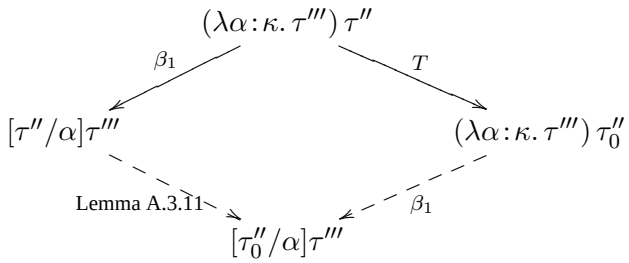


Alternatively, C_2' must be of the form $\lambda\alpha{:}\kappa.\,C_2''$, where $C_2''\,[\tau_2'] = \tau'''$. Then $\tau''' \mapsto C_2''\,[\tau_2''] \equiv \tau_0'''$, and we have



When $C_1 = [\,]$ and $C_2 = \tau'\,C_2'$, again the only reduction rule that applies for $\tau_1' = \tau'\,\tau''$ is β_1, so $\tau' = \lambda\alpha{:}\kappa.\,\tau'''$ for some α, κ, and τ'''. This time, regardless of the structure of C_2', we have that

$\tau'' = C_2''\,[\tau_2'] \mapsto C_2''\,[\tau_2''] \equiv \tau_0''$, hence

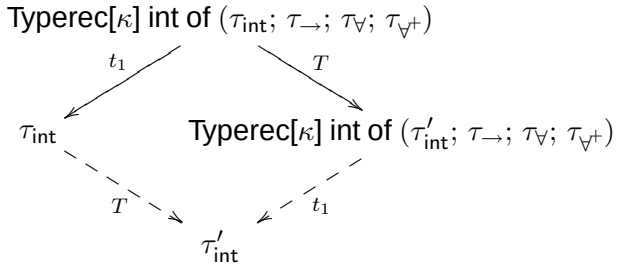


case $\tau = \mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$: The contexts can be empty or of the forms

$$\mathsf{Typerec}[\kappa]\ C\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$$
$$\mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (C;\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$$
$$\mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ C;\ \tau_{\forall};\ \tau_{\forall^+})$$
$$\mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ C;\ \tau_{\forall^+})$$
$$\mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ C)$$

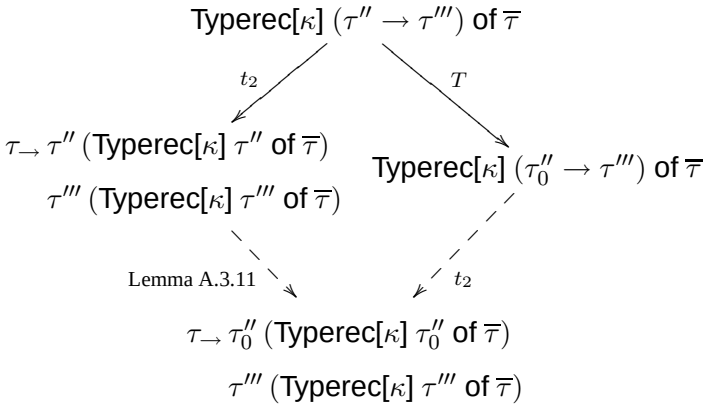
The symmetric cases and the non-overlapping cases are handled as before. Accounting for the symmetry, the remaining cases are when $C_1 = [\,]$ and C_2 is not empty. Then the reduction rule for τ_1' must be one of t_1, t_2, t_3, and t_4. Since there is no η rule for Typerec, the proofs are straightforward.

subcase t_1: then $\tau' = \mathsf{int}$. The result of the reduction under C_2 is ignored and local confluence is trivial, unless $C_2 = \mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (C_2';\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$. In the latter case,



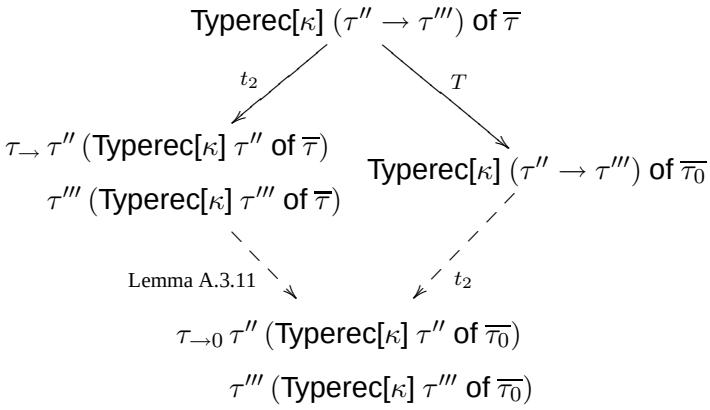
subcase t_2: then $\tau' = \tau'' \rightarrow \tau'''$. We will use $\mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ \overline{\tau}$ as a shorthand for $\mathsf{Typerec}[\kappa]\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})$, and similarly for contexts. If $C_2 = \mathsf{Typerec}[\kappa]\ C_2'\ \mathsf{of}\ \overline{\tau}$, then there are two subcases for C_2' (which must have the $\rightarrow$ constructor at its head). Thus, if

$C_2' = C_2'' \to \tau'''$,



where $\tau'' = C_2''\,[\tau_2'] \mapsto C_2''\,[\tau_2''] \equiv \tau_0''$. The case of $C_2' = \tau'' \to C_2''$ is similar.

Of the other cases we will only show the reduction in the position of $\tau_\to$, writing $\overline{\tau_0}$ for $(\tau_{\mathsf{int}};\ \tau_{\to 0};\ \tau_\forall;\ \tau_{\forall^+})$, where $\tau_\to \mapsto \tau_{\to 0}$.



subcases t_3 and t_4 are similar to t_2. □

Corollary A.3.13 *If $\mathcal{E}; \Delta \vdash \tau : \kappa$, $\tau \mapsto^* \nu$, and $\tau \mapsto^* \tau'$, then $\tau' \mapsto^* \nu$.*

Theorem A.3.14 *If $\mathcal{E}; \Delta \vdash \tau : \kappa$, then there exists exactly one ν such that $\tau \mapsto^* \nu$.*

Proof From Corollaries A.2.24 and A.3.13. □

Appendix B

Formal Properties Of λ_{GC}

In this chapter we prove the soundness of λ_{GC} that we defined in Chapter 4. Throughout this chapter, we assume unique variable names. Our environments are sets with no duplicate occurrences and no ordering. It is easy to show by induction over judgments that extending environments with additional bindings is safe. We will assume this in the rest of the chapter.

Definition B.0.15 *The judgment* $\vdash (M, e)$ *says that the machine state* (M, e) *is well-formed. It is defined by:*

$$\frac{\vdash M : \Psi \qquad \Psi; Dom(\Psi); \cdot; \cdot \vdash e}{\vdash (M, e)}$$

Contrary to the other environments, Ψ is not explicitly constructed in any of the static rules, since it reflects dynamic information. Instead, the soundness proof, or more specifically the type preservation proof, needs to construct some witness Ψ' for the new state (M', e') based on the Ψ of the initial state (M, e).

The code region cd is always implicitly part of the environment. We treat it as a constant region. Even when the environment is restricted to a particular set, say $\Psi|_\Theta$, the code region is included in the restricted set. Therefore $\Psi|_{\nu_1,\ldots\nu_k}$ is equivalent to $\{\mathsf{cd} : \Upsilon_{\mathsf{cd}}, \nu_1 : \Upsilon_{\nu_1}, \ldots \nu_k : \Upsilon_{\nu_k}\}$. And $\Psi|_{\mathsf{cd}}$ is equivalent to $\{\mathsf{cd} : \Upsilon_{\mathsf{cd}}\}$.

Lemma B.0.16 *If* $\Theta', r; \Delta \vdash \sigma$*, then* $[\nu/r]\Theta; \Delta \vdash [\nu/r]\sigma$ *where* $\Theta', r = \Theta$.

Proof The proof is a straightforward induction over the structure of σ. □

Lemma B.0.17 $([\nu/r]\Gamma)|_{\Theta,\nu} = [\nu/r](\Gamma|_{\Theta,r})$

Proof The lemma is proved by induction over the structure of Γ. □

Lemma B.0.18 *If* $\Psi; \Theta', r; \Delta; \Gamma \vdash op \ : \ \sigma$*, then*
$\Psi; [\nu/r]\Theta; \Delta; [\nu/r]\Gamma \vdash [\nu/r]op \ : \ [\nu/r]\sigma$ *where* $\Theta', r = \Theta$.

Proof The proof is by induction over the structure of op. Most of the cases follow directly by induction. We will show only the case for type packages.
case $\langle \alpha = \tau_1, v : \sigma_2 \rangle$: We know that

$$\Psi; \Theta', r; \Delta; \Gamma \vdash \langle \alpha = \tau_1, v : \sigma_2 \rangle \ : \ \exists \alpha : \kappa . \sigma_2$$

This implies that $\Delta \vdash \tau_1$ and $\Psi; \Theta', r; \Delta; \Gamma \vdash v \ : \ [\tau_1/\alpha]\sigma_2$. Applying the inductive hypothesis to the derivation for v, we get that

$$\Psi; [\nu/r]\Theta; \Delta; [\nu/r]\Gamma \vdash [\nu/r]v \ : \ [\tau_1/\alpha]([\nu/r]\sigma_2)$$

This leads to the required result. □

Lemma B.0.19 *If* $\Psi; \Theta', r; \Delta; \Gamma \vdash e$*, then* $\Psi; [\nu/r]\Theta; \Delta; [\nu/r]\Gamma \vdash [\nu/r]e$ *where* $\Theta', r = \Theta$

Proof The proof is by induction over the derivation of e. Most of the cases follow directly from the inductive hypothesis. We will consider only one case here.
case only Θ_1 in e: We get that

$$\Psi; \Theta, r; \Delta; \Gamma \vdash \mathsf{only}\ \Theta_1\ \mathsf{in}\ e$$

This implies that

$$\Psi|_{\Theta_1}; \Theta_1, \mathsf{cd}; \Delta; \Gamma|_{\Theta_1} \vdash e$$

and $\Theta_1 \subset \Theta, r$. Suppose $r \notin \Theta_1$. Then r does not occur free in e. Also $[\nu/r]\Theta_1 = \Theta_1$. Let $\Gamma|_{\Theta_1} = \Gamma_1$. Then we have that
$[\nu/r]\Gamma|_{\Theta_1} = \Gamma_1, \Gamma_2$ and $Dom(\Gamma_1) \cap Dom(\Gamma_2) = \emptyset$. Since we can extend environments, we get that
$\Psi|_{\Theta_1}; \Theta_1, \mathsf{cd}; \Delta; \Gamma_1, \Gamma_2 \vdash e$. Also $\Theta_1 \subset [\nu/r](\Theta, r)$. This leads to the required result.

Suppose now that $r \in \Theta_1$. Suppose that $\Theta_1 = \Theta_2, r$. Then $[\nu/r]\Theta_1 = \Theta_2, \nu$. Then we have that

$$\Psi|_{\Theta_2,r}; \Theta_2, r, \mathsf{cd}; \Delta; \Gamma|_{\Theta_2,r} \vdash e$$

Applying the inductive hypothesis we get that

$$\Psi|_{\Theta_2,r}; [\nu/r]\Theta_2, \mathsf{cd}; \Delta; [\nu/r]\Gamma|_{\Theta_2,r} \vdash [\nu/r]e$$

Applying lemma B.0.17 we get that

$$\Psi|_{\Theta_2,r}; [\nu/r]\Theta_1, \mathsf{cd}; \Delta; [\nu/r]\Gamma|_{\Theta_2,\nu} \vdash [\nu/r]e$$

But we have that $\Psi|_{\Theta_2,r} = \Psi|_{\Theta_2}$. Moreover, $\Psi|_{\Theta_2,\nu} = \Psi|_{\Theta_2}, \Psi'$. Therefore, we get that

$$\Psi|_{\Theta_2,\nu}; [\nu/r]\Theta_1, \mathsf{cd}; \Delta; [\nu/r]\Gamma|_{\Theta_2,\nu} \vdash [\nu/r]e$$

We also have that $[\nu/r]\Theta_1 \subset [\nu/r]\Theta$. This leads to the required result. □

Lemma B.0.20 *If* $\Delta, \alpha{:}\kappa' \vdash \tau \ : \ \kappa$ *and* $\Delta \vdash \tau' \ : \ \kappa'$*, then* $\Delta \vdash [\tau'/\alpha]\tau \ : \ \kappa$

Proof The proof is a straightforward induction over the structure of τ. □

Lemma B.0.21 *If* $\Theta; \Delta, \alpha{:}\kappa \vdash \sigma$ *and* $\Delta \vdash \tau \ : \ \kappa$*, then* $\Theta; \Delta \vdash [\tau/\alpha]\sigma$

Proof The proof is again a straighforward induction over the structure of σ. □

Lemma B.0.22 *If* $\Psi; \Theta; \Delta, \alpha : \kappa; \Gamma \vdash op \ : \ \sigma$ *and* $\Delta \vdash \tau \ : \ \kappa$ *then*
$\Psi; \Theta; \Delta; [\tau/\alpha]\Gamma \vdash [\tau/\alpha]op \ : \ [\tau/\alpha]\sigma$

Proof The proof is a straightforward induction over the structure of op. The only unusual case is when $op = \nu.\ell$. In this case, $\Psi(\nu.\ell) = \sigma$ and $Dom(\Psi); \cdot; \cdot \vdash \sigma$. Therefore, the variable α does not occur free in σ at ν. □

Lemma B.0.23 *If* $\Psi; \Theta; \Delta, \alpha : \kappa; \Gamma \vdash e$ *and* $\cdot \vdash \tau' \ : \ \kappa$ *then* $\Psi; \Theta; \Delta; [\tau'/\alpha]\Gamma \vdash [\tau'/\alpha]e$

Proof The proof is a straightforward induction over the structure of e. The only interesting case is for a typecase when the substituted variable is being analyzed.
case typecase α of $(e_{\mathsf{i}}; e_{\rightarrow}; \alpha_1\alpha_2.e_{\times}; \alpha_e.e_{\exists})$: Suppose we substitute the type τ' for the variable α. Then τ' can only be one of Int, $\tau'' \rightarrow 0$, $\tau'_1 \times \tau'_2$, or $\exists\alpha.\tau''$. For a Int, we need to prove that

$$\Psi; \Theta; \Delta; [\mathsf{Int}/\alpha]\Gamma \vdash [\mathsf{Int}/\alpha](\mathsf{typecase}\ \alpha\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\rightarrow}; \alpha_1\alpha_2.e_{\times}; \alpha_e.e_{\exists}))$$

This implies that we need to prove that

$$\Psi; \Theta; \Delta; [\mathsf{Int}/\alpha]\Gamma \vdash [\mathsf{Int}/\alpha]e_i$$

By definition, we know that

$$\Psi; \Theta; \Delta, \alpha : \Omega; [\mathsf{Int}/\alpha]\Gamma \vdash [\mathsf{Int}/\alpha]e_i$$

Since α is being substituted away, this leads to the required result.

For a code type we need to prove that

$$\Psi; \Theta; \Delta; [\tau' \to 0/\alpha]\Gamma \vdash [\tau' \to 0/\alpha](\mathsf{typecase}\ \alpha\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\to}; \alpha_1\alpha_2.e_{\times}; \alpha_e.e_{\exists}))$$

This implies that we need to prove that

$$\Psi; \Theta; \Delta; [\tau' \to 0/\alpha]\Gamma \vdash [\tau' \to 0/\alpha]e_{\to}$$

By definition, we get that $\Psi; \Theta; \Delta, \alpha : \Omega; \Gamma \vdash e_{\to}$. Substituting for α and applying the inductive hypothesis leads to the result.

For the pair type we need to prove that

$$\Psi; \Theta; \Delta; [(\tau_1' \times \tau_2')/\alpha]\Gamma \vdash [(\tau_1' \times \tau_2')/\alpha](\mathsf{typecase}\ \alpha\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\to}; \alpha_1\alpha_2.e_{\times}; \alpha_e.e_{\exists}))$$

This implies that we need to prove that

$$\Psi; \Theta; \Delta; [(\tau_1' \times \tau_2')/\alpha]\Gamma \vdash [(\tau_1' \times \tau_2'), \tau_1', \tau_2'/\alpha, \alpha_1, \alpha_2]e_{\times}$$

By definition, we know that

$$\Psi; \Theta; \Delta, \alpha : \Omega, \alpha_1 : \Omega, \alpha_2 : \Omega; [\alpha_1 \times \alpha_2/\alpha]\Gamma \vdash [\alpha_1 \times \alpha_2/\alpha]e_{\times}$$

Note that the variables α_1 and α_2 do not occur free separately in Γ. Substituting τ_1' for α_1, τ_2' for α_2, and $\tau_1' \times \tau_2'$ for $\alpha_1 \times \alpha_2$ leads to the required result.

For the existential type we need to prove that

$$\Psi; \Theta; \Delta; [\exists\alpha_1.\tau'/\alpha]\Gamma \vdash [\exists\alpha_1.\tau'/\alpha](\mathsf{typecase}\ \alpha\ \mathsf{of}\ (e_{\mathsf{i}}; e_{\to}; \alpha_1\alpha_2.e_{\times}; \alpha_e.e_{\exists}))$$

This implies that we need to prove that

$$\Psi; \Theta; \Delta; [\exists\alpha_1.\tau'/\alpha]\Gamma \vdash [\exists\alpha_1.\tau', \lambda\alpha_1 : \Omega.\, \tau'/\alpha, \alpha_e]e_{\exists}$$

By definition we know that

$$\Psi; \Theta; \Delta, \alpha : \Omega, \alpha_e : \Omega \to \Omega; [\exists\alpha_1.\alpha_e\ \alpha_1/\alpha]\Gamma \vdash [\exists\alpha_1.\alpha_e\ \alpha_1/\alpha]e_{\exists}$$

Substituting $(\lambda\alpha_1 : \Omega.\, \tau')$ for α_e and applying the inductive hypothesis we get that

$$\Psi; \Theta; \Delta, \alpha{:}\Omega; [\exists \alpha_1.\tau'/\alpha]\Gamma \vdash [\exists \alpha_1.\tau', \lambda \alpha_1{:}\Omega.\, \tau'/\alpha, \alpha_e]e_\exists$$

Since α is being substituted away, we can remove it from the type environment. This leads to the required result. □

Lemma B.0.24 *If* $\Psi; \Theta; \Delta; \Gamma, x : \sigma' \vdash op \ : \ \sigma$ *and* $\Psi; \Theta; \Delta; \Gamma \vdash v' \ : \ \sigma'$ *then*
$\Psi; \Theta; \Delta; \Gamma \vdash [v'/x]op \ : \ \sigma$

Proof The proof is a straightforward induction over the typing derivation for op. □

Lemma B.0.25 *If* $\Psi; \Theta; \Delta; \Gamma \vdash v \ : \ \sigma$ *and* $\Theta_1; \Delta \vdash \sigma$ *and* $\Theta_1 \subset \Theta$*, then*
$\Psi|_{\Theta_1}; \Theta_1; \Delta; \Gamma|_{\Theta_1} \vdash v \ : \ \sigma$

Proof The proof is by induction over the derivation for v. Most of the cases follow directly from the inductive hypothesis. We will consider only one case here.
case $\nu.\ell$: We have that $\Psi; \Theta; \Delta; \Gamma \vdash \nu.\ell \ : \ \sigma$ at ν. This implies that $\Psi(\nu.\ell) = \sigma$ and $Dom(\Psi); \cdot \vdash \sigma$ at ν. However, by assumption we also know that $\Theta_1; \Delta \vdash \sigma$ at ν. This implies that $\nu \in \Theta_1$. This implies that $\Psi|_{\Theta_1}(\nu.\ell) = \sigma$. Moreover, we also get that $\Theta_1; \cdot \vdash \sigma$ at ν. Therefore, we get that $Dom(\Psi|_{\Theta_1}); \cdot \vdash \sigma$ at ν. From here we get that $\Psi|_{\Theta_1}; \Theta_1; \Delta; \Gamma|_{\Theta_1} \vdash \nu.\ell \ : \ \sigma$ at ν. □

Lemma B.0.26 *If* $\Psi; \Theta; \Delta; \Gamma, x : \sigma \vdash e$ *and* $\Psi; \Theta; \Delta; \Gamma \vdash v \ : \ \sigma$ *then*
$\Psi; \Theta; \Delta; \Gamma \vdash [v/x]e$

Proof The proof is again a straightforward induction over the structure of e. We will only show the proof for a couple of cases, the rest of them follow similarly.
case only Θ_1 in e: We have that
$\Psi; \Theta; \Delta; \Gamma, x : \sigma \vdash$ only Θ_1 in e. This implies that
$\Psi|_{\Theta_1}; \Theta_1; \Delta; (\Gamma, x : \sigma)|_{\Theta_1} \vdash e$. If we have that
$\Theta_1; \Delta \vdash \sigma$, then we get that
$\Psi|_{\Theta_1}; \Theta_1; \Delta; \Gamma|_{\Theta_1}, x : \sigma \vdash e$. By lemma B.0.25 we get that
$\Psi|_{\Theta_1}; \Theta_1; \Delta; \Gamma|_{\Theta_1} \vdash v \ : \ \sigma$. Applying the inductive hypothesis gives us that
$\Psi|_{\Theta_1}; \Theta_1; \Delta; \Gamma|_{\Theta_1} \vdash [v/x]e$.

In the other case, we get that
$\Psi|_{\Theta_1}; \Theta_1; \Delta; \Gamma|_{\Theta_1} \vdash e$. This implies that x does not occur free in e. The required result follows from here.

case typecase α of $(e_{\mathsf{i}}; e_{\rightarrow}; \alpha_1\alpha_2.e_{\times}; \alpha_e.e_{\exists})$: By assumption, we get that

$\Delta \vdash \alpha \ : \ \Omega$

$\Psi; \Theta; \Delta; [\mathsf{Int}/\alpha]\Gamma, x : [\mathsf{Int}/\alpha]\sigma \vdash [\mathsf{Int}/\alpha]e_i$

$\Psi; \Theta; \Delta; \Gamma, x : \sigma \vdash e_{\rightarrow}$

$\Psi; \Theta; \Delta, \alpha_1 : \Omega, \alpha_2 : \Omega; [\alpha_1 \times \alpha_2/\alpha]\Gamma, x : [\alpha_1 \times \alpha_2/\alpha]\sigma \vdash [\alpha_1 \times \alpha_2/\alpha]e_{\times}$

$\Psi; \Theta; \Delta, \alpha_e : \Omega \rightarrow \Omega; [\exists\alpha_1.\alpha_e\,\alpha_1/\alpha]\Gamma, x : [\exists\alpha_1.\alpha_e\,\alpha_1/\alpha]\sigma \vdash [\exists\alpha_1.\alpha_e\,\alpha_1/\alpha]e_{\exists}$

By lemma B.0.22, we know that if $\Psi; \Theta; \Delta; \Gamma \vdash v \ : \ \sigma$, then

$\Psi; \Theta; \Delta; [\tau/\alpha]\Gamma \vdash [\tau/\alpha]v \ : \ [\tau/\alpha]\sigma$. Now substitute $[\mathsf{Int}/\alpha]v$ in the e_i branch, substitute v in the $e_{\rightarrow}$ branch, substitute $[\alpha_1 \times \alpha_2/\alpha]v$ in the $e_{\times}$ branch, and substitute $[\exists\alpha_1.\alpha_e\,\alpha_1/\alpha]v$ in the $e_{\exists}$ branch. The required result follows from the inductive hypothesis on each branch. □

Proposition B.0.27 (Type Preservation) *If* $\vdash (M, e)$ *and* $(M, e) \rightsquigarrow (M', e')$ *then* $\vdash (M', e')$.

Proof The proof is by induction over the evaluation relation. We will consider only the cases that do not follow directly from the inductive hypothesis and the substitution lemmas.

case $\nu.\ell[\vec{\tau}][\vec{\nu}](\vec{v})$: The lemma follows from the fact that tag reduction is strongly normalizing and confluent, and preserves kind.

case $\nu.\ell[\vec{\tau'}][\vec{\nu}](\vec{v})$: By definition,

$$\Psi; Dom(\Psi); \cdot; \cdot \vdash \nu.\ell[\vec{\tau'}][\vec{\nu}](\vec{v})$$

Since $M(\nu.\ell) = (\lambda[\alpha \vec{:} \kappa][\vec{r}](\vec{x} : \vec{\sigma}).e)$, we have that

$$\Psi; Dom(\Psi); \cdot; \cdot \vdash \nu.\ell \ : \ \forall[\alpha \vec{:} \kappa][\vec{r}](\vec{\sigma}) \rightarrow 0 \ \mathsf{at} \ \nu$$

This implies that

$$\Psi|_{\mathsf{cd}}; \mathsf{cd}, \vec{r}; \overrightarrow{\alpha : \kappa}; \overrightarrow{x : \sigma} \vdash e$$

By the typing rule, we get that

$$\Psi; Dom(\Psi); \cdot; \cdot \vdash v_i \ : \ [\vec{\nu}, \vec{\tau'}/\vec{r}, \vec{\alpha}]\sigma_i$$

and $\cdot \vdash \tau_i' \ : \ \kappa_i$. From lemma B.0.19 we get that

$$\Psi|_{\mathsf{cd}}; \mathsf{cd}, \vec{\nu}; \Delta; \overrightarrow{x : [\vec{\nu}/\vec{r}]\sigma} \vdash [\vec{\nu}/\vec{r}]e$$

From lemma B.0.23 we get that

$$\Psi|_{\mathsf{cd}}; \mathsf{cd}, \vec{\nu}; \cdot; \overrightarrow{x : [\vec{\nu}, \vec{\tau'}/\vec{r}, \vec{\alpha}]\sigma} \vdash [\vec{\nu}, \vec{\tau'}/\vec{r}, \vec{\alpha}]e$$

Since $\Psi|_{\mathsf{cd}} \subset \Psi$ and $\mathsf{cd}, \vec{\nu} \subset Dom(\Psi)$, we can extend the environment for deriving e. Applying lemma B.0.26 we get that

$$\Psi; Dom(\Psi); \cdot; \cdot \vdash e[\vec{\nu}, \vec{\tau'}, \vec{v}/\vec{r}, \vec{t}, \vec{x}]$$

which leads to the result.

case let $x = \mathsf{put}[\nu]v$ in e: By definition,

$$\Psi; Dom(\Psi); \cdot; \cdot \vdash \mathsf{let}\ x = \mathsf{put}[\nu]v\ \mathsf{in}\ e$$

From the typing rules,

$$\Psi; Dom(\Psi); \cdot; \cdot \vdash \mathsf{put}[\nu]v\ :\ \sigma\ \mathsf{at}\ \nu$$

for some type σ, and $\nu \in Dom(\Psi)$. This implies that

$$\Psi; Dom(\Psi); \cdot; \cdot \vdash v\ :\ \sigma$$

Again, from the typing rules,

$$\Psi, \nu.\ell : \sigma; Dom(\Psi); \cdot; \cdot \vdash \nu.\ell\ :\ \sigma\ \mathsf{at}\ \nu$$

The required result now follows from lemma B.0.26.

case let $x = \mathsf{get}\ \nu.\ell$ in e: By definition,

$$\Psi; Dom(\Psi); \cdot; \cdot \vdash \mathsf{let}\ x = \mathsf{get}\ \nu.\ell\ \mathsf{in}\ e$$

From the typing rules we get that

$$\Psi; Dom(\Psi); \cdot; \cdot \vdash \nu.\ell\ :\ \sigma\ \mathsf{at}\ \nu$$

for some type σ Again from the typing rules, we get that $\Psi(\nu.\ell) = \sigma$. This implies that if $M(\nu.\ell) = v$, then

$$\Psi; Dom(\Psi); \cdot; \cdot \vdash v\ :\ \sigma$$

The required result follows from lemma B.0.26.

case let region r in e: By definition,

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{let\ region}\ r\ \mathsf{in}\ e$$

This implies that

$$\Psi; Dom(\Psi), r; \cdot; \cdot; \cdot \vdash e$$

By lemma B.0.19,

$$\Psi; Dom(\Psi), \nu; \cdot; \cdot; \cdot \vdash [\nu/r]e$$

Since ν is a newly introduced region, we can extend Ψ with it. This implies that

$$\Psi, \nu \mapsto \{\}; Dom(\Psi), \nu; \cdot; \cdot; \cdot \vdash [\nu/r]e$$

This is the required result.

case only Θ in e: By definition,

$$\Psi; Dom(\Psi); \cdot; \cdot; \cdot \vdash \mathsf{only}\ \Theta\ \mathsf{in}\ e$$

This implies that

$$\Psi|_\Theta; \mathsf{cd}, \Theta; \cdot; \cdot; \cdot \vdash e$$

But $\mathsf{cd}, \Theta = Dom(\Psi|_\Theta)$. This implies that

$$\Psi|_\Theta; Dom(\Psi|_\Theta); \cdot; \cdot; \cdot \vdash e$$

which is the required result.

For all of the typecases, the required result follows directly from the typing rules since the value environment is empty.

Lemma B.0.28 (Canonical forms)

1. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v\ :\ \mathsf{int}$ *then* $v = n$.
2. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v\ :\ \sigma\ \mathsf{at}\ \nu$ *then* $v = \nu.\ell$.
3. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v\ :\ \sigma_1 \times \sigma_2$ *then* $v = (v_1, v_2)$.
4. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v\ :\ \exists \alpha : \kappa.\sigma$ *then* $v = \langle \alpha = \tau, v' : \sigma \rangle$.
5. *If* $\Psi; \Theta; \cdot; \cdot; \cdot \vdash v\ :\ \forall [\alpha \vec{:} \kappa][\vec{r}](\vec{\sigma}) \to 0$ *then* $v = \lambda [\alpha \vec{:} \kappa][\vec{r}](x \vec{:} \sigma).e$.

Proof The proof follows from the inspection of the typing rules for values. □

Proposition B.0.29 (Progress) *If* $\vdash (M, e)$ *then either* $e = \mathsf{halt}\ v$ *or there exists a* (M', e') *such that* $(M, e) \rightsquigarrow (M', e')$.

Proof The proof is again by induction over the structure of e. By definition, $\Psi; Dom(\Psi); \cdot; \cdot \vdash e$. The proof for the individual cases start from this point.

case $v[\vec{r}][\vec{\nu}](\vec{v})$: From the typing rules, $v : \forall[\alpha \vec{:} \kappa][\vec{r}](\vec{\sigma}) \to 0$ at ν By lemma B.0.28, $v = \nu.\ell$. From the typing rules

$M(\nu.\ell) = \lambda[\alpha \vec{:} \kappa][\vec{r}](x \vec{:} \sigma).e$ This implies that we have a reduction.

case let $x = op$ in e: If $op = v$, then we have a reduction. If $op = \pi_i v$, then from the typing rules, $\Psi; Dom(\Psi); \cdot; \cdot \vdash v : \sigma_1 \times \sigma_2$. The required result follows from lemma B.0.28. In the case of put$[\nu]v$, the result follows directly. The constraint $\nu \in \Theta$ ensures that $\nu \in Dom(\Psi)$. In the case for get v, by the typing rules we know that $v = \nu.\ell$ for some $\nu.\ell$. Again from the typing rule we know that $\Psi(\nu.\ell) = \sigma$. This implies that $M(\nu.\ell) = v'$ for some value v'.

For the other cases of e, the proposition follows directly from the operational semantics. □

Appendix C

Formal Properties of λ^{i}_{CC}

In this chapter we prove the meta-theoretic properties of our type language λ^{i}_{CC}. The proofs are based on the methods in Werner [Wer94]. We use the formalization of the language presented in Section 5.4. In Section C.1 we prove subject reduction, in Section C.2 we prove the strong normalization, in Section C.3 we prove the Church-Rosser property, in Section C.4 we prove the consistency of the underlying logic.

C.1 Subject reduction

The proof is structured as follows:

- We first define a calculus of unmarked terms. These are terms with no annotations at lambda abstractions. We show that this language is confluent.
- We then prove Geuvers' lemma – a weak form of confluence. It says that a term that is equal to one in head normal form can be reduced to an η-expanded version of this head normal form.
- From Geuvers' lemma, we are able to prove the inversion lemma which relates the structure of a term to its typing derivation.
- We are then able to prove the uniqueness of types and subject reduction for $\beta\iota$ reduction.
- We are then able to prove that the system preserves sorts – that is, if two terms are convertible and well sorted, then they have the same sort.

- Finally, we prove the strengthening lemma and then subject reduction for η reduction.

C.1.1 Unmarked terms

The PTS language is non-confluent. Nederpelt gave the following counterexample – let A be the term defined by $\lambda X : A_1 .\, (\lambda Y : A_2 .\, Y) X$. Then we have that $A \rhd_\beta \lambda X : A_1 .\, X$ and $A \rhd_\eta \lambda Y : A_2 .\, Y$. For our proofs we want to operate in a language that is confluent. We will therefore introduce the notion of unmarked terms. As non-confluence is due to the presence of type annotations in λ abstractions, the unmarked terms are obtained by erasing the type annotations.

The set of unmarked terms $\| A \|$ are defined below. We are given a marked variable $\_$ that can not be used elsewhere.

$$
\begin{aligned}
\| s \| &= s \\
\| X \| &= X \\
\| A_1 \ A_2 \| &= \| A_1 \| \ \| A_2 \| \\
\| \lambda X : A_1 .\, A_2 \| &= \lambda X : \_ .\, \| A_2 \| \\
\| \Pi X : A_1 .\, A_2 \| &= \Pi X : \| A_1 \| \ .\, \| A_2 \| \\
\| \mathsf{Ind}(X : \mathsf{Kind})\{\vec{A}\} \| &= \mathsf{Ind}(X : \mathsf{Kind})\{\overrightarrow{\| A \|}\} \\
\| \mathsf{Ctor}\,(i, A_1) \| &= \mathsf{Ctor}\,(i, \| A_1 \|) \\
\| \mathsf{Elim}[I, A_2](A_1)\{\vec{A}\} \| &= \mathsf{Elim}[\| I \|, \| A_2 \|](\| A_1 \|)\{\overrightarrow{\| A \|}\}
\end{aligned}
$$

Lemma C.1.1 *For all terms A, B, A', B', and for all variables X and Y, we have that $[\lambda Y : A' .\, B/X]A =_{\beta\eta\iota} [\lambda Y : B' .\, B/X]A$.*

Proof Consider $A_2 = [\lambda Z : A' .\, (\lambda Y : B' .\, B)\ Z/X]A$. Then $A_2 \rhd_\beta [(\lambda Z : A' .\, [Z/Y]B)/X]A$ and $A_2 \rhd_\eta [\lambda Y : B' .\, B/X]A$. Alpha converting the first reduct leads to the required result. □

Lemma C.1.2 *For all terms A, we have $A =_{\beta\eta} \| A \|$.*

Proof Follows from lemma C.1.1. □

Definition C.1.3 (ι_0 reduction) *We say that $A \rhd_{\iota_0} \ \| A' \|$ iff $A \rhd_\iota A'$ and $\| A \| \neq \| A' \|$.*

Proposition C.1.4 *For all terms A and A', if $A \rhd_\beta A'$, then $\| A \| \rhd_\beta \| A' \|$ or $\| A \| = \| A' \|$. Similarly, if we have that $A \rhd_\iota A'$, then $\| A \| \rhd_{\iota_0} \| A' \|$ or $\| A \| = \| A' \|$. Moreover, if $\| A \| \rhd_{\beta\iota_0} \| A' \|$, then there exists a A'' such that $A \rhd_{\beta\iota} A''$ and $\| A'' \| = \| A' \|$.*

Lemma C.1.5 (Confluence for unmarked terms) *For all unmarked terms $\| A \|$, the $\beta\eta\iota_0$ reduction is confluent.*

The proof is based on the method of parallel reductions due to Tait and Martin-Löf.

Definition C.1.6 (Parallel reduction) *Define $\twoheadrightarrow$ on unmarked terms as below, in which we assume that $A \twoheadrightarrow A'$, $B \twoheadrightarrow B'$, etc:*

$$
\begin{array}{rl}
A & \twoheadrightarrow A \\
A\ B & \twoheadrightarrow A'\ B' \\
\lambda X : \_.\ A & \twoheadrightarrow \lambda X : \_.\ A' \\
\Pi X : A.\ B & \twoheadrightarrow \Pi X : A'.\ B' \\
\mathsf{Ind}(X : \mathsf{Kind})\{\vec{A}\} & \twoheadrightarrow \mathsf{Ind}(X : \mathsf{Kind})\{\vec{A'}\} \\
\mathsf{Ctor}\,(i, I) & \twoheadrightarrow \mathsf{Ctor}\,(i, I') \\
\mathsf{Elim}[A, C](I)\{\vec{A}\} & \twoheadrightarrow \mathsf{Elim}[A', C'](I')\{\vec{A'}\} \\
(\lambda X : \_.\ A)\ B & \twoheadrightarrow [B'/X]A' \\
\lambda X : \_.\ A\ X & \twoheadrightarrow A' \textit{ if } X \notin FV(A) \\
\mathsf{Elim}[I, C]((\mathsf{Ctor}\,(i, I)\ \vec{B}))\{\vec{A}\} & \twoheadrightarrow (\Phi_{X,I',B'}(C'_i, A'_i))\ \vec{B'} \\
\textit{where}\quad I & = \mathsf{Ind}(X : \mathsf{Kind})\{\vec{C}\} \\
B' & = \lambda Y : \_.\ (\mathsf{Elim}[I', C'](Y)\{\vec{A'}\})
\end{array}
$$

The parallel reduction commutes with respect to substitution.

Lemma C.1.7 *If $A \twoheadrightarrow A'$ and $B \twoheadrightarrow B'$, then $[B/X]A \twoheadrightarrow [B'/X]A'$.*

Proof By induction over the fact that $A \twoheadrightarrow A'$. □

The parallel reduction also has the following properties with respect to terms such as products and inductive definitions. The proof in each case is immediate and follows by induction over the structure of the term.

Proposition C.1.8 *Suppose $A = \Pi X : \vec{B}.\, Y\ \vec{C}$. If A can be reduced to A' through a reduction relation ($\twoheadrightarrow$, $\rhd_\beta$, etc.), then $A' = \Pi X : \vec{B'}.\, Y\ \vec{C'}$ where all the $\vec{B}$ and $\vec{C}$ can be reduced to $\vec{B'}$ and $\vec{C'}$ by the same reduction relation.*

Proposition C.1.9 *Suppose $A = \Pi X : \vec{B}.\, Y\ \vec{C}$ and $A' = \Pi X : \vec{B'}.\, Y\ \vec{C'}$ be two terms such that both can be reduced to A'' through a reduction relation ($\twoheadrightarrow$, $\rhd_\beta$, etc.). Then $A'' = \Pi X : \vec{B''}.\, Y\ \vec{C''}$ where $\vec{B}$ and $\vec{B'}$ can be reduced to $\vec{B''}$ by the same relation and $\vec{C}$ and $\vec{C'}$ can be reduced to $\vec{C''}$ by the same relation.*

The parallel reduction is important because it subsumes the single step reduction; that is, if $A \rhd A'$, then we have that $A \twoheadrightarrow A'$ which also implies that $A \rhd^* A'$. From here, to show the confluence of $\rhd$, it suffices to show the confluence of parallel reduction.

Lemma C.1.10 *For all unmarked terms D, D', D'', we have that if $D \twoheadrightarrow D'$ and $D \twoheadrightarrow D''$, then there exists a D''' such that $D' \twoheadrightarrow D'''$ and $D'' \twoheadrightarrow D'''$.*

Proof The proof is by induction over the structure of D. We will only show one case here.

- Suppose $D = \mathsf{Elim}[I, C]((\mathsf{Ctor}\ (i, I)\ \vec{B}))\{\vec{A}\}$.
  - We can then have $D' = (\Phi_{X,I',B'}(C'_i, A'_i))\ \vec{B'}$ and $D'' = (\Phi_{X,I'',B'}(C''_i, A''_i))\ \vec{B''}$. We have that $I' = \mathsf{Ind}(X : \mathsf{Kind})\{\vec{C'}\}$ and $I'' = \mathsf{Ind}(X : \mathsf{Kind})\{\vec{C''}\}$. This implies that $C_i \twoheadrightarrow C'_i$ and $C_i \twoheadrightarrow C''_i$. By applying the induction hypothesis to the subterms, we get that $I' \twoheadrightarrow I'''$ and $I'' \twoheadrightarrow I'''$ and so on for the other subterms. From here and proposition C.1.9, it follows that we can take $D''' = (\Phi_{X,I''',B'}(C'''_i, A'''_i))\ \vec{B'''}$.
  - Suppose $D' = \mathsf{Elim}[I', C']((\mathsf{Ctor}\ (i, I')\ \vec{B'}))\{\vec{A'}\}$ and $D'' = (\Phi_{X,I'',B'}(C''_i, A''_i))\ \vec{B''}$. As above we can again define I''', C'''_i, etc. and take $D''' = (\Phi_{X,I''',B'}(C'''_i, A'''_i))\ \vec{B'''}$.
  - Also $D' = \mathsf{Elim}[I', C']((\mathsf{Ctor}\ (i, I')\ \vec{B'}))\{\vec{A'}\}$ and $D'' = \mathsf{Elim}[I'', C'']((\mathsf{Ctor}\ (i, I'')\ \vec{B''}))\{\vec{A''}\}$. In this case, we can again take that $D''' = \mathsf{Elim}[I''', C''']((\mathsf{Ctor}\ (i, I''')\ \vec{B'''}))\{\vec{A'''}\}$.

□

As a corollary of the confluence of unmarked terms we get the following:

Corollary C.1.11 *If A and B are two distinct sorts or two distinct variables or a variable and a sort, then we have that $A \neq B$.*

We will need another lemma – that of the delay of η reduction. But before that, we have to define another variant of the ι reduction. This essentially says that a ι reduction that would appear

only after a series of eta reductions can be reduced straightaway without going through the eta reductions. For well typed terms, this is equivalent to ι reduction, but it also allows us to retain the property of delay of η reduction for ill-typed terms.

$$\begin{array}{l}\mathsf{Elim}[I, A''](\lambda\vec{X}:\vec{A}'.\,(\mathsf{Ctor}\,(i,I)\ \vec{A})\vec{C}')\{\vec{B}\}\ \ \triangleright_{\iota'}\ \ (\Phi_{X,I,B'}(C_i, B_i))\ \vec{A} \\ \textit{where } I = \mathsf{Ind}(X:\mathsf{Kind})\{\vec{C}\} \\ \qquad B' = \lambda Y:I.\,(\mathsf{Elim}[I, A''](Y)\{\vec{B}\}) \\ \qquad C_i' \triangleright_\eta X_i \text{ and } X_i \notin FV(\vec{A}) \cup FV(I)\end{array}$$

Proposition C.1.12 *For all terms A_1 and A_2, we have that $A_1 =_{\beta\eta\iota} A_2$ if and only if $A_1 =_{\beta\eta\iota'} A_2$.*

Lemma C.1.13 *If $A \triangleright_\eta A' \ \triangleright_{\beta\iota'}\ A''$, then either $A \triangleright^*_{\beta\iota'} A''$, or there exists a A''' such that $A \ \triangleright_{\beta\iota'}\ A''' \triangleright^*_\eta A''$.*

Proof The proof is by induction over the structure of A. We will consider only the cases that do not follow directly from the induction hypothesis.

- $A = C\ D$. There are two cases.
  - If $C \triangleright_\eta C'$, then it follows immediately from the induction hypothesis.
  - If $D \triangleright_\eta D'$ and $C = \lambda X:B.\,B'$ and $A'' = [D'/X]B'$, then take $A''' = [D/X]B'$. The other cases follow from the induction hypothesis.
- $A = \lambda X : C.\,B\ X$. Suppose $A'' = B'$ where $B\ \triangleright_{\beta\iota'}\ B'$. But then we also have that $A\ \triangleright_{\beta\iota'}\ \lambda X:C.\,B'\ X$. Since the reduction does not introduce new free variables, this term can now η-reduce to B'.

□

Lemma C.1.14 (Delay of η reduction) *For all terms A and A', if $A \triangleright^* A'$, then there exists a term A'' such that $A \triangleright^*_{\beta\iota'} A'' \triangleright^*_\eta A'$.*

Proof Follows from lemma C.1.13. □

We will next prove Geuvers' lemma which is essentially a weak form of confluence. This is enough to prove the uniqueness of types and subject reduction. But before that we need to define the counterpart of the ι' reduction for unmarked terms. We define it in the obvious way

Definition C.1.15 (ι_0' reduction) *We say that* $A \rhd_{\iota_0'} \| A' \|$ *iff* $A \rhd_{\iota'} A'$ *and* $\| A \| \neq \| A' \|$.

As before it has the following property:

Proposition C.1.16 *Suppose* $A \rhd_{\iota'} A'$. *Then either* $\| A \| = \| A' \|$, *or* $\| A \| \rhd_{\iota_0'} \| A' \|$. *Moreover, if* $\| A \| \rhd_{\iota_0'} \| A' \|$, *then* $A \rhd_{\iota'} A'$.

Lemma C.1.17 (Geuvers lemma)

- If $A =_{\beta\eta\iota} X\ \vec{A}$, then $A \rhd^*_{\beta\iota'} \lambda \vec{Y} : \vec{A'}.\ (X\ \vec{B}\ \vec{C})$ where for all i, $A_i =_{\beta\eta\iota} B_i$ and for all j, $C_j \rhd^*_\eta Y_j$.
- If $A =_{\beta\eta\iota} \Pi X : A_1.\ A_2$, then $A \rhd^*_{\beta\iota'} \lambda \vec{Y} : \vec{A'}.\ ((\Pi X : A_3.\ A_4)\ \vec{B})$ where $A_1 =_{\beta\eta\iota} A_3$ and $A_2 =_{\beta\eta\iota} A_4$ and for all i, $B_i \rhd^*_\eta Y_i$.
- If $A =_{\beta\eta\iota} \mathsf{Ctor}\ (i, I)\ \vec{C}$, then $A \rhd^*_{\beta\iota'} \lambda \vec{Y} : \vec{A'}.\ ((\mathsf{Ctor}\ (i, I')\ \vec{C'})\ \vec{B})$ where for all i, $C_i =_{\beta\eta\iota} C_i'$ and for all j, $B_j \rhd^*_\eta Y_j$, and $I =_{\beta\eta\iota} I'$.
- If $A =_{\beta\eta\iota} \mathsf{Ind}(X : \mathsf{Kind})\{\vec{A}\}\ \vec{C}$, then $A \rhd^*_{\beta\iota'} \lambda \vec{Y} : \vec{A'}.\ ((\mathsf{Ind}(X : \mathsf{Kind})\{\vec{A''}\})\ \vec{C'})\ \vec{B}$ where for all i, $A_i =_{\beta\eta\iota} A_i''$ and for all j, $C_j =_{\beta\eta\iota} C_j'$, and for all k, $B_k \rhd^*_\eta Y_k$.
- If $A =_{\beta\eta\iota} \mathsf{Elim}[I, A_2](A_1)\{\vec{A''}\}\ \vec{C}$, then $A \rhd^*_{\beta\iota'} \lambda \vec{Y} : \vec{A'}.\ (\mathsf{Elim}[I', B'](B)\{\vec{B}\}\ \vec{C'})\ \vec{B'}$ where $A_1 =_{\beta\eta\iota} B$, and $A_2 =_{\beta\eta\iota} B'$, and $I =_{\beta\eta\iota} I'$, and for all i, $A_i'' =_{\beta\eta\iota} B_i$ and for all j, $C_j =_{\beta\eta\iota} C_j'$ and for all k, $B_k' \rhd^*_\eta Y_k$.

Proof The proof for each of the cases is similar and is by induction over the length of the equivalence relation. We will show only one case here.

- Suppose $A =_{\beta\eta\iota} X\ \vec{A}$. By the induction hypothesis, there exists an A'' such that $A'' \rhd^*_{\beta\iota'} \lambda \vec{Y} : \vec{A'}.\ (X\ \vec{B}\ \vec{C})$ and $A \rhd_{\beta\iota} A''$ or $A'' \rhd_{\beta\iota} A$.
  - The case where $A \rhd_{\beta\iota} A''$ is immediate.
  - The case where $A \rhd_\eta A''$ follows from the lemma of delay of η-reduction.
  - If $A'' \rhd_{\beta\iota} A$, then the required result follows from the confluence of $\beta\iota'$ reduction.
  - Suppose $A'' \rhd_\eta A$. Then from the confluence of $\beta\eta\iota_0'$ reduction on unmarked terms, we get that $\| A \| \rhd^* X\ \vec{D}$ where $\| B_i \| \rhd_{\beta\eta\iota_0'} D_i$. From the lemma of delay of η-reduction, we get that

$$\| A \| \rhd^*_{\beta\iota_0'} \lambda \vec{Y} : \_.\ X\ \vec{D'} \vec{F} \rhd^*_\eta X\ \vec{D}$$

From proposition C.1.16 we can deduce the existence of a term A_1 such that $A \rhd^*_{\beta\iota'} A_1$ and $\| A_1 \| = \lambda \vec{Y} : \_. \; X \; \vec{D'} \vec{F}$. The required result follows from here.

□

C.1.2 Classification of terms

Definition C.1.18 *We partition the set of terms into four classes: the set of types* Ty*, the set of kinds* Ki*, the set of kind schemas* Sc*, and* Ex*. The class of a term is defined as follows:*

$$\begin{aligned}
\mathsf{Cls}(\mathsf{Kind}) &= \mathsf{Sc} \\
\mathsf{Cls}(\mathsf{Kscm}) &= \mathsf{Ex} \\
\mathsf{Cls}(\alpha) &= \mathsf{Ty} \\
\mathsf{Cls}(j) &= \mathsf{Ki} \\
\mathsf{Cls}(z) &= \mathsf{Sc} \\
\mathsf{Cls}(A_1 \; A_2) &= \mathsf{Cls}(A_1) \\
\mathsf{Cls}(\lambda X : A_1 . \; A_2) &= \mathsf{Cls}(A_2) \\
\mathsf{Cls}(\Pi X : A_1 . \; A_2) &= \mathsf{Cls}(A_2) \\
\mathsf{Cls}(\mathsf{Ind}(X : \mathsf{Kind})\{\vec{A}\}) &= \mathsf{Ki} \\
\mathsf{Cls}(\mathsf{Ctor}\,(i, A_1)) &= \mathsf{Ty} \\
\mathsf{Cls}(\mathsf{Elim}[I, A_2](A_1)\{\vec{A}\}) &= \mathsf{Ty} \quad \text{if } \mathsf{Cls}(A_2) = \mathsf{Ki}\text{, else } \mathsf{Ki}
\end{aligned}$$

We also define the following function:

$$\begin{aligned}
\mathsf{lift}(\mathsf{Ty}) &= \mathsf{Ki} \\
\mathsf{lift}(\mathsf{Ki}) &= \mathsf{Sc} \\
\mathsf{lift}(\mathsf{Sc}) &= \mathsf{Ex}
\end{aligned}$$

Lemma C.1.19 *If* $\Delta \vdash A_1 : A_2$ *is derivable, then we have* $\mathsf{lift}(\mathsf{Cls}(A_1)) = \mathsf{Cls}(A_2)$*. In particular,* $A_1 \neq \mathsf{Ext}$*. Moreover, for all pairs* (X, A) *in* Δ*, we have* $\mathsf{Cls}(A) = \mathsf{lift}(\mathsf{Cls}(X))$*.*

Proof Immediate by induction over the derivation of the judgment. □

C.1.3 Well typed terms

We now consider the well typed terms. The following lemmas are proved easily by induction over the typing derivations.

Lemma C.1.20 (Substitution) *If we can derive* $\Delta_1, (X, A), \Delta_2 \vdash B : C$ *and* $\Delta_1 \vdash A_2 : A$*, then we can derive* $\Delta_1, ([A_2/X]\Delta_2) \vdash [A_2/X]B : [A_2/X]C$.

Proof Straightforward induction over the structure of the derivation. □

Lemma C.1.21 *If we can derive* $\Delta_1, (X, A), \Delta_2 \vdash B : C$*, then we also have that* $\Delta_1 \vdash A : s$ *for some sort* s*. Moreover, we also have that* $\Delta_1, (X, A), \Delta_2 \vdash A : s$.

Proof The proof is by induction over the structure of the derivation. □

Lemma C.1.22 *If we have that* $\Delta \vdash \Pi X : A.\, B : s$*, then we have that* $\Delta, X : A \vdash B : s$.

Proof The only interesting case is for the CONV case which follows from Corollary C.1.11. □

Lemma C.1.23 *If the judgment* $\Delta \vdash A : B$ *is derivable, then either* $B = \mathsf{Ext}$*, or* $\Delta \vdash B : s$ *for some sort* s.

Proof The proof is a straightforward induction over the structure of the derivation. □

Lemma C.1.24 (Inversion) *If the judgment* $\Delta \vdash A : B$ *is derivable, then*

$$
\begin{array}{lll}
A = \alpha & \Rightarrow & \alpha \in \Delta, \quad B =_{\beta\eta\iota} \Delta(\alpha), \quad \Delta \vdash B : \mathsf{Kind} \\
A = j & \Rightarrow & j \in \Delta, \quad B =_{\beta\eta\iota} \Delta(j), \quad \Delta \vdash B : \mathsf{Kscm} \\
A = z & \Rightarrow & z \in \Delta, \quad B =_{\beta\eta\iota} \Delta(z), \quad \Delta \vdash B : \mathsf{Ext} \\
A = \mathsf{Kind} & \Rightarrow & B =_{\beta\eta\iota} \mathsf{Kscm} \\
A = \mathsf{Kscm} & \Rightarrow & B = \mathsf{Ext} \\
A = \Pi X\!:\!A_1.\, A_2 & \Rightarrow & \Delta \vdash A_1 : s_1, \quad \Delta, X\!:\!A_1 \vdash A_2 : s_2, \quad B =_{\beta\eta\iota} s_2 \\
 & & \textit{where } s_1 \textit{ is any sort and} \\
 & & s_2 = \mathsf{Kind}, \textit{ or}, \quad s_1 \in \{\mathsf{Kind}, \mathsf{Kscm}\} \textit{ and} \quad s_2 = \mathsf{Kscm} \\
A = \lambda X\!:\!A_1.\, A_2 & \Rightarrow & \Delta \vdash A_1 : s_1, \quad \Delta, X\!:\!A_1 \vdash A_2 : A_3, \quad \Delta \vdash A_3 : s_2 \\
 & & B =_{\beta\eta\iota} \Pi X\!:\!A_1.\, A_3, \quad \Delta \vdash B : s_2 \\
A = A_1\ A_2 & \Rightarrow & \Delta \vdash A_1 : \Pi X\!:\!B'.\, A', \quad \Delta \vdash A_2 : B', \quad B =_{\beta\eta\iota} [A_2/X]A' \\
A = \mathsf{Ind}(X\!:\!\mathsf{Kind})\{\vec{A}\} & \Rightarrow & \Delta, X\!:\!\mathsf{Kind} \vdash A_i : \mathsf{Kind}, \quad wfc_X(A_i), \quad B =_{\beta\eta\iota} \mathsf{Kind} \\
A = \mathsf{Ctor}\,(i, I) & \Rightarrow & I = \mathsf{Ind}(X\!:\!\mathsf{Kind})\{\vec{A}\}, \quad \textit{same conditions on } I, \quad B =_{\beta\eta\iota} [I/X]A_i \\
A = \mathsf{Elim}[I, A'](A)\{\vec{B}\} & \Rightarrow & I = \mathsf{Ind}(X\!:\!\mathsf{Kind})\{\vec{A}\}, \quad \textit{same conditions on } I \\
 & & \Delta \vdash A : I, \quad \Delta \vdash A' : I \to \mathsf{Kind}, \quad \Delta \vdash B : \mathsf{Kind} \\
 & & B =_{\beta\eta\iota} A'\ A, \quad \Delta \vdash B_i : \zeta_{X,I}(A_i, A', \mathsf{Ctor}\,(i, I)) \\
A = \mathsf{Elim}[I, A'](A)\{\vec{B}\} & \Rightarrow & I = \mathsf{Ind}(X\!:\!\mathsf{Kind})\{\vec{A}\}, \quad \textit{same conditions on } I \\
 & & \Delta \vdash A : I, \quad \Delta \vdash A' : \mathsf{Kscm} \\
 & & \Delta \vdash B : \mathsf{Kscm} \textit{ and } B =_{\beta\eta\iota} A' \\
 & & \Delta \vdash B_i : \Psi_{X,I}(A_i, A'), \quad \textit{for all } i \ \textit{ small}(A_i)
\end{array}
$$

Proof By induction over the structure of the derivation. For every case we consider the set of possible typing derivations. □

Lemma C.1.25 (Uniqueness of types) *If* $\Delta \vdash A : A_1$ *and* $\Delta \vdash A : A_2$*, then* $A_1 =_{\beta\eta\iota} A_2$.

Proof By induction over the structure of A. We use the fact that if $A_1 =_{\beta\eta\iota} B$ and $A_2 =_{\beta\eta\iota} B$, then $A_1 =_{\beta\eta\iota} A_2$. For every case, we use the corresponding clause from lemma C.1.24. □

Corollary C.1.26 *Suppose A is a well typed term. If $A \rhd_{\iota'} A'$, then $A \rhd_{\iota} A'$.*

C.1.4 Reductions on well typed terms

Lemma C.1.27 (Subject reduction for $\beta\iota$ reduction) *If the judgment $\Delta \vdash A : B$ is derivable, and if $A \rhd_{\beta\iota} A'$ and $\Delta \rhd_{\beta\iota} \Delta'$, then we have that*

$$\Delta \vdash A' : B \qquad \Delta' \vdash A : B$$

Proof The interesting cases are the APP and ELIM.

- APP When only the sub-terms reduce without a reduction at the head, the lemma follows by using the induction hypothesis on the sub-terms. Suppose that

$$A = \lambda X\!:\!A_1.\, A_2 \qquad \frac{\Delta \vdash A : \Pi X\!:\!B'.\, A' \quad \Delta \vdash B : B'}{\Delta \vdash A\ B : [B/X]A'}$$

  and $A\ B \rhd_{\beta} [B/X]A_2$. We know from lemma C.1.24 that

$$\begin{array}{l} \Delta, X\!:\!A_1 \vdash A_2 : A_3 \\ \Pi X\!:\!A_1.\, A_3 =_{\beta\eta\iota} \Pi X\!:\!B'.\, A' \\ \Delta \vdash A_1 : s_1 \\ \Delta \vdash B' : s_2 \end{array}$$

  This implies that $A_1 =_{\beta\eta\iota} B'$ and $A_3 =_{\beta\eta\iota} A'$. Moreover,

$$\mathsf{Cls}(B') = \mathsf{Cls}(A_1) = \mathsf{lift}(\mathsf{Cls}(X))$$

  Therefore, we get from lemma C.1.19 that

$$\mathsf{Cls}(s_2) = \mathsf{Cls}(s_1) \quad \Rightarrow s_2 = s_1$$

  Applying the CONV rule we get that $\Delta \vdash B : A_1$. By lemma C.1.20 we get that $\Delta \vdash$

$[B/X]A_2 : [B/X]A_3$. We can show in a similar manner as before that $\mathsf{Cls}(A_3) = \mathsf{Cls}(A')$. This allows us to apply the CONV rule again which leads to the required result.

- L-ELIM We will only consider the case when an ι reduction takes place at the head. The other cases follow easily by structural induction.

$$\frac{\Delta \vdash A : I \qquad \Delta \vdash A' : \mathsf{Kscm} \qquad \textit{for all } i \qquad \Delta \vdash B_i : \Psi_{X,I}(C_i, A')}{\Delta \vdash \mathsf{Elim}[I, A'](A)\{\vec{B}\} : A'}$$
$$\textit{where } I = \mathsf{Ind}(X : \mathsf{Kind})\{\vec{C}\} \textit{and } \forall i.\, small(C_i)$$

The interesting case is when we consider the reduction

$$\begin{aligned} &\mathsf{Elim}[I, A'](\mathsf{Ctor}\ (i, I)\ \vec{A})\{\vec{B}\} \ \rhd_\iota\ (\Phi_{X,I,B'}(C_i, B_i))\ \vec{A} \\ &\textit{where} \quad I = \mathsf{Ind}(X : \mathsf{Kind})\{\vec{C}\} \\ &\qquad B' = \lambda Y : I.\, (\mathsf{Elim}[I, A'](Y)\{\vec{B}\}) \end{aligned}$$

Suppose $A'' = (\Phi_{X,I,B'}(C_i, B_i))\ \vec{A}$. Suppose that $\vec{A} = A_{1\ldots n}$. We have that $\Delta \vdash B_i : \Psi_{X,I}(C_i, A')$. The proof is by induction on the fact that C_i is a kind of a constructor and the length of $\vec{A}$. We consider the different cases by which C_i is a kind of a constructor.

  - If $C_i = X$, then $A'' = B_i$. From definition 5.4.6 we can see that in this case, B_i has the type A'.
  - If $C_i = \Pi Y : B.\, C$, then
    $A'' = (\Phi_{X,I,B'}([A_1/Y]C, B_i\ A_1))\ A_{2\ldots n}$. We have that $\Delta \vdash B_i\ A_1 : \Psi_{X,I}([A_1/Y]C, A')$. By the induction hypothesis, the reduct has type A'.
  - If $C_i = \Pi \vec{Y} : \vec{B}.\, X \rightarrow C$, then

$$A'' = \Phi_{X,I,B'}(C, B_i\ A_1\ (\lambda \vec{Y} : \vec{B}.\, B'\ (A_1\ \vec{Y})))\ A_{2\ldots n}$$

    From Definition 5.4.6 we have that
    $\Delta \vdash B_i : [I/X]A \rightarrow [A'/X]A \rightarrow \Psi_{X,I}(C', A')$. We also know that $\Delta \vdash A_1 : [I/X]A$. From here, we can apply the induction hypothesis and show that the reduct has type A'.

- ELIM We will only consider the case when an ι reduction takes place at the head. The other cases follow easily by structural induction.

$$\frac{\begin{array}{c}\Delta \vdash A : I \qquad \Delta \vdash A' : I \to \mathsf{Kind} \\ \textit{for all } i \qquad \Delta \vdash B_i : \zeta_{X,I}(C_i, A', \mathsf{Ctor}\,(i, I))\end{array}}{\begin{array}{c}\Delta \vdash \mathsf{Elim}[I, A'](A)\{\vec{B}\} : A'\ A\text{`} \\ \textit{where } I = \mathsf{Ind}(X : \mathsf{Kind})\{\vec{C}\}\end{array}}$$

The interesting case is when we consider the reduction

$$\begin{array}{l}\mathsf{Elim}[I, A'](\mathsf{Ctor}\,(i, I)\ \vec{A})\{\vec{B}\} \ \rhd_\iota \ (\Phi_{X,I,B'}(C_i, B_i))\ \vec{A} \\ \textit{where} \quad I = \mathsf{Ind}(X : \mathsf{Kind})\{\vec{C}\} \\ \qquad B' = \lambda Y : I.\, (\mathsf{Elim}[I, A'](Y)\{\vec{B}\})\end{array}$$

Suppose $A'' = (\Phi_{X,I,B'}(C_i, B_i))\ \vec{A}$. Suppose that $\vec{A} = A_{1\ldots n}$. We have that $\Delta \vdash B_i : \zeta_{X,I}(C_i, A', \mathsf{Ctor}\,(i, I))$. By using the inversion lemma we can get that $\Delta \vdash B' : \Pi X : I.\, A'\ X$. By induction on the structure of C_i (where C_i is a kind of a constructor), we can show that if $C_i = \Pi \vec{Y} : \vec{B}.\, X$, then $\Delta \vdash \Phi_{X,I,B'}(C_i, B_i) : \Pi \vec{Y} : \vec{B}.\, A'\ \mathsf{Ctor}\,(i, I)\ \vec{Y}$. The required result follows from here.

□

Corollary C.1.28 *Suppose A is a well formed term. If $A \rhd^*_{\beta\iota'} A'$, then $A \rhd^*_{\beta\iota} A'$ and A' is well formed.*

Corollary C.1.29 *Suppose A is a well formed term. If $A \rhd^* A'$, then there exists a well formed term A'' such that $A \rhd^*_{\beta\iota} A'' \rhd^*_\eta A'$.*

Lemma C.1.30 *Let $\Delta \vdash A : B$ and $\Delta \vdash A' : B'$ be two derivable judgments. If $A =_{\beta\eta\iota} A'$, then* $\mathsf{Cls}(A) = \mathsf{Cls}(A')$.

Proof We know that $\| A \|$ and $\| A' \|$ have a common reduct, say A_2. This implies that

$$\| A \| \rhd^*_{\beta\iota'_0} B \rhd^*_\eta A_2 \qquad \text{and} \qquad \| A' \| \rhd^*_{\beta\iota'_0} B' \rhd^*_\eta A_2$$

From here we get that

$$A \rhd^*_{\beta\iota} B_0 \text{ and } A' \rhd^*_{\beta\iota} B'_0 \text{ where } \| B_0 \| = B \text{ and } \| B'_0 \| = B'$$

Eta reduction does not change the class of a term. Moving from marked to unmarked terms also does not change the class of a term. Therefore, we get that

$$\begin{aligned} &\mathsf{Cls}(A) = \mathsf{Cls}(B_0) = \mathsf{Cls}(B) = \mathsf{Cls}(A_2) \quad \text{and} \\ &\mathsf{Cls}(A_2) = \mathsf{Cls}(B') = \mathsf{Cls}(B'_0) = \mathsf{Cls}(A') \end{aligned}$$

□

Corollary C.1.31 *Let $\Delta \vdash A : s_1$ and $\Delta \vdash B : s_2$ be two derivable judgments. If $A =_{\beta\eta\iota} B$, then $s_1 = s_2$.*

Lemma C.1.32 *If $\Delta_1, Y : C, \Delta_2 \vdash A : B$ and $Y \notin FV(\Delta_2) \cup FV(A)$, then there exists a B' such that $\Delta_1\Delta_2 \vdash A : B'$. (This also implies that $B =_{\beta\eta\iota} B'$).*

Proof The proof is by induction on the structure of the derivation. We will consider only the important cases.

- case FUN. We know that

$$\frac{\Delta_1, Y : C, \Delta_2, X : A \vdash B : B' \qquad \Delta_1, Y : C, \Delta_2 \vdash \Pi X : A.\, B' : s}{\Delta_1, Y : C, \Delta_2 \vdash \lambda X : A.\, B : \Pi X : A.\, B'}$$

  Applying the induction hypothesis to the formation of B

$$\Delta_1\Delta_2, X : A \vdash B : C' \qquad B' =_{\beta\eta\iota} C'$$

  By lemma C.1.23 we have that

$$\Delta_1\Delta_2, X : A \vdash C' : s \text{ which implies } \Delta_1\Delta_2 \vdash \Pi X : A.\, C' : s$$

  Therefore we get that

$$\Delta_1\Delta_2 \vdash \lambda X : A.\, B : \Pi X : A.\, C'$$

- case APP We know that

$$\frac{\Delta_1, Y:C, \Delta_2 \vdash A : \Pi X:B'.\, A' \qquad \Delta_1, Y:C, \Delta_2 \vdash B : B'}{\Delta_1, Y:C, \Delta_2 \vdash A\ B : [B/X]A'}$$

By applying the induction hypothesis we get that

$$\Delta_1\Delta_2 \vdash A : A_2 \text{ and } \Delta_1\Delta_2 \vdash B : A_3 \text{ where } A_2 =_{\beta\eta\iota} \Pi X:B'.\, A' \text{ and } A_3 =_{\beta\eta\iota} B'$$

From lemma C.1.17, $A_2 \rhd_{\beta\iota} \lambda\vec{Y}:\vec{A}.\, (\Pi X:B''.\, A'')\ \vec{B}$. Since $\beta\iota$ reduction preserves type, and A_2 is well formed, we have that $A_2 \rhd_{\beta\iota} \Pi X:B''.\, A''$. This implies that $A'' =_{\beta\eta\iota} A'$ and $B'' =_{\beta\eta\iota} B'$. We also get that $A_3 =_{\beta\eta\iota} B''$. From corollary C.1.31 we get that A_3 and B'' have the same sort. By applying the CONV rule we get that

$$\Delta_1\Delta_2 \vdash A : \Pi X:B''.\, A'' \text{ and } \Delta_1\Delta_2 \vdash B : B''$$

Therefore, we get that

$$\Delta_1\Delta_2 \vdash A\ B : [B/X]A''$$

□

As a corollary we now get that

Lemma C.1.33 (Strengthening) *If* $\Delta_1, Y:C, \Delta_2 \vdash A : B$ *and* $Y \notin FV(\Delta_2) \cup FV(A) \cup FV(B)$, *then* $\Delta_1\Delta_2 \vdash A : B$.

Lemma C.1.34 (Subject reduction for η reduction) *If* $\Delta \vdash A : B$, *and* $A \rhd_\eta A'$ *and* $\Delta \rhd_\eta \Delta'$, *then we have that*

$$\Delta \vdash A' : B \qquad \Delta' \vdash A : B$$

Proof The interesting case is that of functions. Suppose that

$$\Delta \vdash \lambda X:A_1.\, A_2\ X : B \quad X \notin FV(A_2) \quad \lambda X:A_1.\, A_2\ X \rhd_\eta A_2$$

From lemma C.1.24 we know that

$$\Delta, X : A_1 \vdash A_2\ X : A_3 \quad B =_{\beta\eta\iota} \Pi X : A_1.\, A_3 \quad \Delta \vdash B : s$$

Again applying lemma C.1.24 we get that

$$\Delta, X : A_1 \vdash A_2 : \Pi Y : B'.\, A' \ \ B' =_{\beta\eta\iota} A_1 \ \ A_3 =_{\beta\eta\iota} [X/Y]A'$$

By applying the CONV rule now, we get that $\Delta, X : A_1 \vdash A_2 : B$. By applying lemma C.1.33 we get that $\Delta \vdash A_2 : B$. □

Theorem C.1.35 (Subject reduction) *If $\Delta \vdash A : B$, and $A \rhd A'$ and $\Delta \rhd \Delta'$, then we have that: $\Delta \vdash A' : B$ and $\Delta' \vdash A : B$.*

Proof Follows from lemma C.1.27 and C.1.34. □

C.2 Strong normalization

The proof is structured as follows:

- We introduce a calculus of pure terms. This is just the pure λ calculus extended with a recursive filtering operator. We do this so that we can operate in a confluent calculus.
- We define a notion of reducibility candidates. Every schema gives rise to a reducibility candidate. We also show how these candidates can be constructed inductively.
- We then define a notion of well constructed kinds which is a weak form of typing.
- We associate an interpretation to each well formed kind. We show that under adequate conditions, this interpretation is a candidate.
- We show that type level constructs such as abstractions and constructors belong to the candidate associated with their kind.
- We show that the interpretation of a kind remains the same under $\beta\eta$ reduction.
- We define a notion of kinds that are invariant on their domain – these are kinds whose interpretation remains the same upon reduction.

- We show that kinds formed with large elimination are invariant on their domain.
- From here we can show the strong normalization of the calculus of pure terms. We show that if a type is well formed, then the pure term derived from it is strongly normalizing.
- We then reduce the strong normalization of all well formed terms to the strong normalization of pure terms.

C.2.1 Notation

The syntax for the language is:

$$
\begin{array}{lll}
(\textit{ctxt}) & \Delta & ::= \cdot \mid \Delta, X : A \\
(\textit{sort}) & s & ::= \mathsf{Kind} \mid \mathsf{Kscm} \mid \mathsf{Ext} \\
(\textit{var}) & X & ::= z \mid j \mid \alpha \\
(\textit{ptm}) & A, B & ::= s \mid X \mid \lambda X : A.\, B \mid A\ B \mid \Pi X : A.\, B \mid \mathsf{Ind}(X : \mathsf{Kind})\{\vec{A}\} \\
 & & \quad \mid \mathsf{Ctor}\,(i, A) \mid \mathsf{Elim}[A', B'](A)\{\vec{B}\}
\end{array}
$$

The proof of strong normalization uses the stratification in the language shown below.

$$
\begin{array}{lll}
(\textit{ctxt}) & \Delta & ::= \cdot \mid \Delta, z : \mathsf{Kscm} \mid \Delta, j : u \mid \Delta, \alpha : \kappa \\
(\textit{kscm}) & u & ::= z \mid \Pi \alpha : \kappa.\, u \mid \Pi j : u_1.\, u_2 \mid \mathsf{Kind} \\
(\textit{kind}) & \kappa & ::= j \mid \lambda \alpha : \kappa_1.\, \kappa_2 \mid \kappa[\tau] \mid \lambda j : u.\, \kappa \mid \kappa_1\, \kappa_2 \mid \Pi \alpha : \kappa_1.\, \kappa_2 \mid \Pi j : u.\, \kappa \\
 & & \quad \mid \Pi z : \mathsf{Kscm}.\, \kappa \mid \mathsf{Ind}(j : \mathsf{Kind})\{\vec{\kappa}\} \mid \mathsf{Elim}[\kappa', u](\tau)\{\vec{\kappa}\} \\
(\textit{type}) & \tau & ::= \alpha \mid \lambda \alpha : \kappa.\, \tau \mid \tau_1\, \tau_2 \mid \lambda j : u.\, \tau \mid \tau[\kappa] \mid \lambda z : \mathsf{Kscm}.\, \tau \mid \tau[u] \\
 & & \quad \mid \mathsf{Ctor}\,(i, \kappa) \mid \mathsf{Elim}[\kappa', \kappa](\tau')\{\vec{\tau}\}
\end{array}
$$

In this section, the types are also referred to as proof terms. We sometimes use I to refer to an inductive definition.

C.2.2 Pure terms

The pure terms are defined as:

$$
(\Lambda) \quad a, b, c ::= \alpha \mid a\ b \mid \lambda \alpha.a \mid \mathsf{Co}(n) \mid \mathsf{match}\,\alpha.\{\vec{a}\}
$$

The set of reductions on the pure terms are defined as:

$$
\begin{aligned}
&(\lambda\alpha.a)\ b \rhd_\beta [b/\alpha]a \\
&\lambda\alpha.(a\ \alpha) \rhd_\eta a \qquad \text{if } \alpha \notin FV(a) \\
&\mathsf{match}\,\alpha.\{\vec{a}\}\ (\mathsf{Co}(i)\ \vec{b}) \rhd_\iota ([\mathsf{match}\,\alpha.\{\vec{a}\}/\alpha]a_i)\ \vec{b}
\end{aligned}
$$

The translation from types to pure terms is defined as:

$$
\begin{aligned}
|\alpha| &= \alpha \\
|\tau_1\,\tau_2| &= |\tau_1|\ |\tau_2| \\
|\tau[\kappa]| &= |\tau| \\
|\tau[u]| &= |\tau| \\
|\lambda\alpha\!:\!\kappa.\,\tau| &= \lambda\alpha.|\tau| \\
|\lambda j\!:\!u.\,\tau| &= |\tau| \\
|\lambda z\!:\!\mathsf{Kscm}.\,\tau| &= |\tau| \\
|\mathsf{Ctor}\,(n,\kappa)| &= \mathsf{Co}(n) \\
|\mathsf{Elim}[\kappa,\kappa'](\tau)\{\vec{\tau}\}| &= (\mathsf{match}\,\alpha.\{\overrightarrow{\Upsilon(\kappa_i, |\tau_i|, \lambda\alpha_2.\alpha\ \alpha_2)}\})\ \ |\tau| \\
&\text{where } \kappa = \mathsf{Ind}(j\!:\!\mathsf{Kind})\{\vec{\kappa}\} \text{ and}
\end{aligned}
$$

$$
\begin{aligned}
\Upsilon(j, a_1, a_2) &= a_1 \\
\Upsilon(\Pi\alpha\!:\!\kappa_1.\,\kappa_2, a_1, a_2) &= \lambda\alpha.\Upsilon(\kappa_2, a_1\ \alpha, a_2) \\
\Upsilon(\Pi j\!:\!u.\,\kappa, a_1, a_2) &= \Upsilon(\kappa, a_1, a_2) \\
\Upsilon(\Pi z\!:\!\mathsf{Kscm}.\,\kappa, a_1, a_2) &= \Upsilon(\kappa, a_1, a_2) \\
\Upsilon(\Pi\vec{X}\!:\!\vec{A}.\,j \to \kappa, a_1, a_2) &= \lambda\alpha.\Upsilon(\kappa, a_1\ \alpha\ (\lambda|\vec{X}|.a_2\ (\alpha\ \ |\vec{X}|)), a_2)
\end{aligned}
$$

Lemma C.2.1 *Let τ and τ' be two well formed types and let α be a type variable. Then* $|[\tau'/\alpha]\tau| = [|\tau'|\ /\alpha]\ |\tau|$.

Proof It is a straightforward proof by induction over the structure of τ. □

The following lemma uses Definitions 5.4.6 and 5.4.4 in Section C.1 and also the definition of Υ from above.

Lemma C.2.2 $|\Phi_{X,I,B}(\kappa,\tau)| = [\mathsf{match}\,\alpha.\{\overrightarrow{\Upsilon(\kappa_i, |\tau_i|, \lambda\alpha_2.\alpha\ \alpha_2)}\}/\alpha]\Upsilon(\kappa, |\tau|, \lambda\alpha_2.\alpha\ \alpha_2)$

Proof The proof is by induction on the fact that κ is the kind of a constructor. □

Lemma C.2.3 *For all well formed proof terms τ_1 and τ_2, if $\tau_1 \ \rhd^i \ \tau_2$, then $|\,\tau_1\,| \ \rhd^j \ |\,\tau_2\,|$ where $j \leq i$.*

Proof Follows from lemmas C.2.1 and C.2.2. □

C.2.3 Interpretation of schemas

Definition C.2.4 (Arity) *We call ground kind schemas arities denoted as* $\mathsf{arity}(u, \mathsf{Kind})$. *The arities are defined with the following grammar:*

$$(\textit{kscm}) \quad u ::= \mathsf{Kind} \mid \Pi j : u_1 .\, u_2 \mid \Pi \alpha : \kappa .\, u$$

Definition C.2.5 (Schema map) *We define a kind schema mapping $\mathcal{K}$ as a function mapping kind schema variables z to arities. We also use $\mathcal{K}, z : u$ to say that $\mathcal{K}$ has been augmented with the mapping $z \mapsto u$.*

Definition C.2.6 *We define the function $\rho(u)_{\mathcal{K}}$ as:*

$$\rho(u)_{\mathcal{K}} = \rho_0(\mathcal{K}(u)) \quad \textit{where}$$

- $\rho_0(\mathsf{Kind})$ *is the set of sets of pure terms;*
- $\rho_0(\Pi j : u_1 .\, u_2)$ *is the set of functions from $\rho_0(u_1)$ to $\rho_0(u_2)$; and*
- $\rho_0(\Pi \alpha : \kappa .\, u)$ *is the set of functions from Λ to $\rho_0(u)$.*

Definition C.2.7 *For each kind schema u and mapping $\mathcal{K}$, we define in $\rho(u)_{\mathcal{K}}$ the relation of partial equivalence written as $\simeq_{\mathcal{K}(u)}$ as follows:*

- *for all C and C' in $\rho_0(\mathsf{Kind})$, we have that $C \simeq_{\mathsf{Kind}} C' \iff C = C'$;*
- *for all C and C' in $\rho_0(\Pi j : u_1 .\, u_2)$, we have $C \simeq_{\Pi j u_1 .\, u_2} C' \iff$ for all C_1 and C_2 in $\rho_0(u_1)$ with $C_1 \simeq_{u_1} C_2$ we get that $C\ C_1 \simeq_{u_2} C'\ C_2$; and*
- *for all C and C' in $\rho_0(\Pi \alpha : \kappa .\, u)$, we have that $C \simeq_{\Pi \alpha \kappa .\, u} C' \iff$ for all a and b in Λ such that $a =_{\beta\eta\iota} b$, we get that $C\ a \simeq_u C' b$.*

Definition C.2.8 (Invariant) *Given C in $\rho(u)_{\mathcal{K}}$, we say that C is invariant $\iff C \simeq_{\mathcal{K}(u)} C$.*

Definition C.2.9 (Neutral terms) *A term is called neutral if it has neither of the following forms – $\lambda\alpha.a$,* $\mathsf{Co}(i)\ \vec{a}$*, or* $\mathsf{match}\ \alpha.\{\vec{a}\}$.

Definition C.2.10 *We define $\mathcal{CR}_0(\mathsf{Kind})$ as consisting of all sets C such that:*

- *if $a \in C$, then a is strongly normalizing;*
- *if $a_1 \rhd a_2$ and $a_1 \in C$, then $a_2 \in C$; and*
- *if a is neutral and for all terms a' such that $a \rhd a'$ and $a' \in C$, then $a \in C$.*

Definition C.2.11 (Candidates) *We define $\mathcal{CR}(u)_{\mathcal{K}}$ as a subset of $\rho(u)_{\mathcal{K}}$ as:*

$$\mathcal{CR}(u)_{\mathcal{K}} = \mathcal{CR}_0(\mathcal{K}(u)) \quad \textit{where}$$

- *$\mathcal{CR}_0(\mathsf{Kind})$ is defined as in Definition C.2.10;*
- *$\mathcal{CR}_0(\Pi\alpha : \kappa.\, u)$ is the set of invariant elements C belonging to $\rho_0(\Pi\alpha : \kappa.\, u)$ such that $C\ \Lambda \subset \mathcal{CR}_0(u)$; and*
- *$\mathcal{CR}_0(\Pi j : u_1.\, u_2)$ is the set of invariant elements C belonging to $\rho_0(\Pi j : u_1.\, u_2)$ such that $C\ (\mathcal{CR}_0(u_1)) \subset \mathcal{CR}_0(u_2)$.*

Proposition C.2.12 *All reducibility candidates are invariant.*

Proposition C.2.13 *Let $(C_i)_{i\in I}$ be a family of reducibility candidates of* Kind *indexed by a set I. Then $\cap_{i\in I} C_i$ is a reducibility candidate of schema* Kind.

Lemma C.2.14 *Let $C \in \rho(u)_{\mathcal{K}}$. If C is invariant, then*

$$C \in \mathcal{CR}(u)_{\mathcal{K}} \iff \forall C' \in Dom(\mathcal{CR}(u)_{\mathcal{K}}).C\ C' \in \mathcal{CR}(\mathsf{Kind})_{\mathcal{K}}$$

Proof Straightforward induction over the structure of $\mathcal{K}(u)$. □

Definition C.2.15 *Let a_1 be a strongly normalizing term. Then the length of the longest sequence of reductions to a normal form is denoted as $\nu(a_1)$.*

Lemma C.2.16 *Let a_1 and a_2 be two terms and let $C \in \mathcal{CR}_0(\mathsf{Kind})$ be a reducibility candidate. If a_2 is strongly normalizing, and if $[a_2/\alpha]a_1 \in C$, then $(\lambda\alpha.a_1)\ a_2 \in C$.*

Proof By induction over $\nu(a_1) + \nu(a_2)$. □

Corollary C.2.17 *Let a_1 be a pure term and let C be a reducibility candidate of schema* Kind*. Let $\vec{\alpha}$ and $\vec{a'}$ be respectively a sequence of variables and terms of the same length. If for all i, a'_i is strongly normalizing, and if $[\vec{a'}/\vec{\alpha}]a_1 \in C$, then $(\lambda\vec{\alpha}.a_1)\ \vec{a'} \in C$.*

Lemma C.2.18 *For all reducibility candidates C of kind* Kind*, for all sequences of strongly normalizing $\vec{a}$ and $\vec{b}$ and for all i less than the length of $\vec{a}$, we have that*

$$\mathsf{match}\ \alpha.\{\vec{a}\}\ (\mathsf{Co}(i)\ \vec{b}) \in C \iff ([\mathsf{match}\ \alpha.\{\vec{a}\}/\alpha]a_i)\ \vec{b} \in C$$

Proof Follows by induction over $\nu(a_i) + \nu(b_i)$ (for all i). □

Definition C.2.19 (Canonical candidates) *Define* $\mathsf{Can}(u)_{\mathcal{K}}$ *as:*

$$\mathsf{Can}(u)_{\mathcal{K}} = \mathsf{Can}_0(\mathcal{K}(u)) \quad \textit{where}$$

- $\mathsf{Can}_0(\mathsf{Kind})$ *is the set of all strongly normalizing terms;*
- $\mathsf{Can}_0(\Pi\alpha:\kappa.\,u)$ *is the function mapping all pure terms to* $\mathsf{Can}_0(u)$*; and*
- $\mathsf{Can}_0(\Pi j:u_1.\,u_2)$ *is the function mapping all elements of $\rho_0(u_1)$ to* $\mathsf{Can}_0(u_2)$.

C.2.4 Properties of candidates

In this section, we state some properties of the reducibility candidates. The properties with respect to the union and the intersection of a family of candidates will be used for the inductive constructions of candidates.

Definition C.2.20 (Order over candidates) *For each kind schema u and mapping $\mathcal{K}$, we define in $\rho(u)_{\mathcal{K}}$ the relation $<_{\mathcal{K}(u)}$ as follows:*

- *for all C and C' in $\rho_0(\mathsf{Kind})$, we have that $C <_{\mathsf{Kind}} C' \iff C \subset C'$;*

- *for all C and C' in $\rho_0(\Pi j : u_1 .\, u_2)$, we have $C <_{\Pi j u_1 .\, u_2} C' \iff$ for all C_1 in $\rho_0(u_1)$, we get that $C\ C_1 <_{u_2} C'\ C_1$; and*

- *for all C and C' in $\rho_0(\Pi \alpha : \kappa .\, u)$, we have that $C <_{\Pi \alpha \kappa .\, u} C' \iff$ for all a in Λ, we get that $C\ a <_u C'\ a$.*

Definition C.2.21 *For all schemas u and mapping $\mathcal{K}$, for all families of elements in $\rho(u)_{\mathcal{K}}$, we define $\bigwedge_{i \in I} C_i$ as:*

- *for all $C_i \in \rho_0(\mathsf{Kind})$, $\bigwedge_{i \in I} C_i = \cap_{i \in I} C_i$;*

- *for all $C_i \in \rho_0(\Pi \alpha : \kappa .\, u)$, $\bigwedge_{i \in I} C_i = b \in \Lambda \mapsto \bigwedge_{i \in I} C_i\ b$; and*

- *for all $C_i \in \rho_0(\Pi j : u_1 .\, u_2)$, $\bigwedge_{i \in I} C_i = C' \in \rho_0(u_1) \mapsto \bigwedge_{i \in I} C_i\ C'$.*

Lemma C.2.22 *Let u be a schema and $\mathcal{K}$ a mapping and C_i a family of elements of $\rho(u)_{\mathcal{K}}$. Then $\forall j \in I$, $\bigwedge_{i \in I} C_i <_{\mathcal{K}(u)} C_j$.*

Proof It follows in a straightforward way by induction over the structure of $\mathcal{K}(u)$. □

The following two propositions also follow easily by induction over the structure of $\mathcal{K}(u)$.

Proposition C.2.23 *Let u be a schema and $\mathcal{K}$ a mapping and C_i a family of elements of $\rho(u)_{\mathcal{K}}$. If all C_i are invariants, then the same holds for $\bigwedge_{i \in I} C_i$.*

Proposition C.2.24 *Let u be a schema and $\mathcal{K}$ a mapping and C_i a family of elements of $\mathcal{CR}(u)_{\mathcal{K}}$. Then we also have that $\bigwedge_{i \in I} C_i \in \mathcal{CR}(u)_{\mathcal{K}}$.*

Corollary C.2.25 *We get that $(\mathcal{CR}(u)_{\mathcal{K}}, <_{\mathcal{K}(u)})$ is an inf-semi-lattice for all schema u and mapping $\mathcal{K}$. We use $min(\mathcal{K}(u))$ to denote the smallest element.*

Definition C.2.26 *For all schemas u and mapping $\mathcal{K}$, for all families of elements in $\rho(u)_{\mathcal{K}}$, we define $\bigvee_{i \in I} C_i$ as:*

- *for all $C_i \in \rho_0(\mathsf{Kind})$, $\bigvee_{i \in I} C_i = \cup_{i \in I} C_i$;*

- *for all $C_i \in \rho_0(\Pi \alpha : \kappa .\, u)$, $\bigvee_{i \in I} C_i = b \in \Lambda \mapsto \bigvee_{i \in I} C_i\ b$; and*

- *for all $C_i \in \rho_0(\Pi j : u_1 .\, u_2)$, $\bigvee_{i \in I} C_i = C' \in \rho_0(u_1) \mapsto \bigvee_{i \in I} C_i\ C'$.*

Lemma C.2.27 *Let u be a schema and $\mathcal{K}$ be a mapping. Let $(C_i)_{i\in I}$ and $(C'i)_{i\in I}$ be two families of elements of $\rho(u)_{\mathcal{K}}$. If for all elements i of I we have that $C_i \simeq_{\mathcal{K}(u)} C'_i$, then we also have that $\bigvee_{i\in I} C_i \simeq_{\mathcal{K}(u)} \bigvee_{i\in I} C'_i$.*

Proof Straightforward induction over the structure of $\mathcal{K}(u)$. □

Corollary C.2.28 *Let u be a schema and $\mathcal{K}$ be a mapping. Let $(C_i)_{i\in I}$ be a family of elements of $\rho(u)_{\mathcal{K}}$. If all C_i are invariant, then $\bigvee_{i\in I} C_i$ is also invariant.*

Lemma C.2.29 *Let u be a schema and $\mathcal{K}$ be a mapping. Let $(C_i)_{i\in I}$ be a family of elements of $\rho(u)_{\mathcal{K}}$ and $C \in \rho(u)_{\mathcal{K}}$. If for all i, $C_i <_{\mathcal{K}(u)} C$, then $\bigvee_{i\in I} C_i <_{\mathcal{K}(u)} C$.*

Proof The proof is by induction over the structure of $\mathcal{K}(u)$. □

Lemma C.2.30 *Let $(C_i)_{i\in I}$ be a totally ordered family of elements of $\mathcal{CR}(u)_{\mathcal{K}}$. Then $\bigvee_{i\in I} C_i \in \mathcal{CR}(u)_{\mathcal{K}}$.*

Proof The proof is by induction over the structure of $\mathcal{K}(u)$. Suppose $\bigvee_{i\in I} C_i = C'$.

- $\mathcal{K}(u) = \mathsf{Kind}$. We have to make sure that all three conditions in Definition C.2.10 are satisfied. The first two conditions follow obviously. For the third case, assume that a is neutral and for all terms a_i such that $a \rhd a_i$, we have that $a_i \in C'$. This implies that $a_i \in C_j$ for some j. Since there are finitely many such C_j and they are totally ordered, we can choose a C_k among them that contains all the C_js. Since this C_k is also a candidate, it contains a. Therefore, $a \in \bigvee_{i\in I} C_i$.

- $\mathcal{K}(u) = \Pi\alpha : \kappa.\, u$. Since all the C_i are invariant, it follows from Definitions C.2.7 and C.2.8 that for a term $a \in \Lambda$, we have that $C_i\ a$ is invariant. Again from Definition C.2.20, it is clear that the $C_i\ a$ are totally ordered. Also from Corollary C.2.28 we get that $\bigvee_{i\in I} C_i\ a$ is invariant. Applying the induction hypothesis we get that $\bigvee_{i\in I} C_i\ a \in \mathcal{CR}_0(u)$. From Definition C.2.11, it follows that $\bigvee_{i\in I} C_i \in \mathcal{CR}_0(\Pi\alpha : \kappa.\, u)$.

- $\mathcal{K}(u) = \Pi j : u_1.\, u_2$. Similar to the previous case.

□

Definition C.2.31 (Schema interpretation) *A schema interpretation $\mathcal{U}$ is a function that maps a kind variable j to an element of $\rho(u)_{\mathcal{K}}$. We also use $\mathcal{U}, j : C$ to say that $\mathcal{U}$ has been augmented with the mapping $j \mapsto C$.*

Definition C.2.32 (Well formed kinds) *Let u be a schema, κ be a kind, $\mathcal{K}$ be a mapping, and $\mathcal{U}$ be an interpretation. We say that κ is a well formed kind of schema $\mathcal{K}(u)$ under $\mathcal{K}$ and $\mathcal{U}$* iff *:*

1. $\kappa = j$ *and* $\mathcal{U}(j) = \rho(u)_{\mathcal{K}}$*;*
2. $\kappa = \Pi\alpha : \kappa_1 . \kappa_2$ *with* $\mathcal{K}(u) =_{\beta\eta\iota}$ Kind *and* κ_1 *and* κ_2 *are both well constructed of schema* Kind *under* $\mathcal{K}$ *and* $\mathcal{U}$*;*
3. $\kappa = \Pi j : u' . \kappa'$ *with* $\mathcal{K}(u) =_{\beta\eta\iota}$ Kind *and* κ' *is well constructed of schema* Kind *under* $\mathcal{K}$ *and* $\mathcal{U}, j : \rho(u')_{\mathcal{K}}$*;*
4. $\kappa = \Pi z : \mathsf{Kscm} . \kappa'$ *with* $\mathcal{K}(u) =_{\beta\eta\iota}$ Kind *and for all* u' *such that* $u' \in \mathsf{arity}(u_1, \mathsf{Kind})$*, we have that* κ' *is well constructed of schema* Kind *under* $\mathcal{K}, z : u'$ *and* $\mathcal{U}$*;*
5. $\kappa = \kappa_1\ \kappa_2$ *if there exists two schemas* u_1 *and* u_2 *with* κ_2 *well constructed of schema* $\mathcal{K}(u_2)$ *under* $\mathcal{K}$ *and* $\mathcal{U}$*, also* κ_1 *well constructed of schema* $\mathcal{K}(\Pi j : u_2 . u_1)$ *under* $\mathcal{K}$ *and* $\mathcal{U}$*, and* $\rho(u)_{\mathcal{K}} = \rho([\kappa_2/j]u_1)_{\mathcal{K}}$*;*
6. $\kappa = \kappa_1\ \tau_1$ *if there exists a schema* u_2 *and kind* κ_2 *such that* κ_1 *is well constructed of schema* $\mathcal{K}(\Pi\alpha : \kappa_2 . u_2)$ *under* $\mathcal{K}$ *and* $\mathcal{U}$ *and* $\rho(u)_{\mathcal{K}} = \rho([\tau_1/\alpha]u_2)_{\mathcal{K}}$*;*
7. $\kappa = \lambda j : u_1 . \kappa_1$ *if there exists a* u_2 *such that* κ_1 *is well constructed of schema* $\mathcal{K}(u_2)$ *under* $\mathcal{K}$ *and* $\mathcal{U}, j : \rho(u_1)_{\mathcal{K}}$ *and* $\rho(u)_{\mathcal{K}} = \rho(\Pi j : u_1 . u_2)_{\mathcal{K}}$*;*
8. $\kappa = \lambda\alpha : \kappa_1 . \kappa_2$ *if there exists a* u_2 *such that* κ_2 *is well constructed of schema* $\mathcal{K}(u_2)$ *under* $\mathcal{K}$ *and* $\mathcal{U}$ *and* $\rho(u)_{\mathcal{K}} = \rho(\Pi\alpha : \kappa_1 . u_2)_{\mathcal{K}}$*;*
9. $\kappa = \mathsf{Ind}(j : \mathsf{Kind})\{\vec{\kappa}\}$ *if all* κ_i *are kinds of constructors and well constructed of schema* Kind *under* $\mathcal{K}$ *and* $\mathcal{U}, j : \rho_0(\mathsf{Kind})$*, and* $\rho(u)_{\mathcal{K}} = \rho_0(\mathsf{Kind})$*; and*
10. $\kappa = \mathsf{Elim}[\kappa', u'](\tau)\{\vec{\kappa}\}$ *if* $\kappa' = \mathsf{Ind}(j : \mathsf{Kind})\{\vec{\kappa'}\}$*, and* κ' *is well constructed of schema* Kind *under* $\mathcal{K}$ *and* $\mathcal{U}$*, also* u' *is a schema and* $\mathcal{K}(u) =_{\beta\eta\iota} u'$*, and* κ_i *is well constructed of schema* $\mathcal{K}(\Psi_{j,\kappa'}(\kappa'_i, u'))$ *under* $\mathcal{K}$ *and* $\mathcal{U}$*.*

Definition C.2.33 *We define compatible mappings and interpretation as:*

1. *A mapping $\mathcal{K}$ is compatible with a context Δ if for all $z \in \Delta$, we have $\mathcal{K}(z) = \mathsf{arity}(u, \mathsf{Kind})$.*
2. *An interpretation $\mathcal{U}$ is compatible with a context Δ and a compatible mapping $\mathcal{K}$ if for all pairs $(j, u) \in \Delta$, we have $\mathcal{U}(j) \in \rho(u)_{\mathcal{K}}$.*

Lemma C.2.34 *If $\Delta \vdash \kappa : u$, then for all compatible $\mathcal{K}$ and $\mathcal{U}$, we have that κ is well constructed of schema $\mathcal{K}(u)$.*

Proof By induction over the structure of κ. □

C.2.5 Inductive constructions

Consider an increasing function F in $\rho_0(\mathsf{Kind})$ for the order $<_{\mathsf{Kind}}$. Denote the smallest element of $\rho_0(\mathsf{Kind})$ as $\bot$. Since $\rho_0(\mathsf{Kind})$ is closed under $\cap$, and $(\rho_0(\mathsf{Kind}), <_{\mathsf{Kind}})$ is an inf-semi-lattice, the function F has a least fixed point (lfp). We will construct this least fixed point inductively. We first define the transfinite iteration of F.

Definition C.2.35 *Let $C \in \rho_0(\mathsf{Kind})$ and o be an ordinal. We define the iteration of order o of F over C as:*

- $F^0(C) = C$;
- $F^{o+1}(C) = F(F^o(C))$; *and*
- $F^{lim(U)} = \cup_{o \in U} F^o(C)$.

Lemma C.2.36 *Let o be an ordinal; we have $F^o(\bot) <_{\mathsf{Kind}} lfp(F)$.*

Proof The proof is by induction over o. If $o = 0$, then it follows immediately. Otherwise,

- $o = o' + 1$ Then we have that $F^o(\bot) = F(F^{o'}(\bot))$. By the induction hypothesis, we get that $F(F^{o'}(\bot)) <_{\mathsf{Kind}} F(lfp(F))$. This implies that $F(F^{o'}(\bot)) <_{\mathsf{Kind}} lfp(F)$.
- $o = lim(U)$ Follows immediately from the induction hypothesis and lemma C.2.29.

□

Remark C.2.37 *Since we do not consider the degenerate case of $F(\perp) = \perp$, it follows from lemma C.2.36 that for some ordinal o, we have that $lfp(F) = F^o(\perp)$.*

Lemma C.2.38 *Suppose $\mathcal{S}$ is a subset of $\rho_0(\mathsf{Kind})$ satisfying:*

- *if $(C_i)_{i \in I}$ is a totally ordered family of elements of $\mathcal{S}$, then $\cup_{i \in I} C_i \in \mathcal{S}$;*
- *$F(\perp) \in \mathcal{S}$; and*
- *for all C in $\mathcal{S}$, $F(C) \in \mathcal{S}$.*

Then $lfp(F) \in \mathcal{S}$.

Proof Follows from the fact that $lfp(F) = F^o(\perp)$ for some ordinal o. □

Definition C.2.39 *Let $a \in lfp(F)$. We define $\mathsf{deg}(a)$ as the smallest ordinal such that $a \in F^{\mathsf{deg}(a)}(\perp)$.*

Definition C.2.40 *To all $a \in lfp(F)$, we associate $\mathsf{pred}(a)$ defined as $F^{\mathsf{deg}(a)-1}(\perp)$.*

Lemma C.2.41 *For all a, $\mathsf{deg}(a)$ is an ordinal successor.*

Proof Suppose it is the limit of the set U. From Definition C.2.35, there exists some $o \in U$ for which $a \in F^o(\perp)$. This leads to a contradiction. □

Definition C.2.42 (Partial order) *Suppose C and C' are two elements of $\mathcal{CR}_0(\mathsf{Kind})$. We say that $C <_F C'$ if $C = F^o(\perp)$ and $C' = F^{o'}(\perp)$, and $o < o'$.*

C.2.6 Interpretation of kinds

In this section we interpret kinds as members of reducibility candidates. First we augment the schema interpretation

Definition C.2.43 *We augment $\mathcal{U}$ so that it maps a kind variable to an element of $\rho(u)_{\mathcal{K}}$, and a type variable to a pure term a.*

Definition C.2.44 *We denote the interpretation of a type τ as $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau)$. To form this, we first construct the corresponding pure term $|\tau|$ and then substitute the type variables by the corresponding pure terms in $\mathcal{U}$. This is equivalent to $\mathcal{U}(|\tau|)$.*

Definition C.2.45 (Interpreting kinds) *Consider a kind κ, a schema u, a mapping $\mathcal{K}$, and an interpretation $\mathcal{U}$. Suppose κ is well constructed of schema $\mathcal{K}(u)$ under $\mathcal{K}$ and $\mathcal{U}$. We define by recursion on κ:*

1. $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(j) = \mathcal{U}(j)$
2. $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\Pi\alpha : \kappa_1 . \kappa_2) = \{a \in \Lambda, \forall a_1 \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_1), a\ a_1 \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U},\alpha:a_1}(\kappa_2)\}$
3. $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\Pi j : u_1 . \kappa_1) = \cap_{C \in \mathcal{CR}(u_1)_{\mathcal{K}}} \mathcal{C}^{\mathcal{K}}_{\mathcal{U},j:C}(\kappa_1)$
4. $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\Pi z : \mathsf{Kscm} . \kappa_1) = \cap_{u_1 \in \mathsf{arity}(u,\mathsf{Kind})} \mathcal{C}^{\mathcal{K},z:u_1}_{\mathcal{U}}(\kappa_1)$
5. $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_1\ \tau) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_1)\ \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau)$
6. $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_1\ \kappa_2) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_1)\ \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_2)$
7. $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\lambda\alpha : \kappa_1 . \kappa_2) = a \in \Lambda \longmapsto \mathcal{C}^{\mathcal{K}}_{\mathcal{U},\alpha:a}(\kappa_2)$
8. $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\lambda j : u_1 . \kappa_1) = C \in \mathcal{CR}(u_1)_{\mathcal{K}} \longmapsto \mathcal{C}^{\mathcal{K}}_{\mathcal{U},j:C}(\kappa_1)$
9. $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\mathsf{Ind}(j : \mathsf{Kind})\{\vec{\kappa}\}) =$ *the least fixed point of the function F from $\rho_0(\mathsf{Kind})$ to $\rho_0(\mathsf{Kind})$ defined as :*

   for all $\mathcal{S} \in \rho_0(\mathsf{Kind})$, for all C' in $\mathcal{CR}(I \to \mathsf{Kind})_{\mathcal{K}}$ (where $I = \mathsf{Ind}(j : \mathsf{Kind})\{\vec{\kappa}\}$), for all sequences of pure terms b_i, with for all i,

$$b_i \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U},j:\mathcal{S},A':C',B':\mathsf{Co}_{(i)}}(\zeta_{j,I}(\kappa_i, A', B'))$$

   $F(\mathcal{S})$ is the union of $min(\mathsf{Kind})$ with the set of pure terms a such that

$$(\mathsf{match}\ \alpha.\{\overrightarrow{\mathcal{C}^{\mathcal{K}}_{\mathcal{U},a_i:b_i}(\Upsilon(\kappa_i, a_i, \lambda\alpha_2.\alpha\ \alpha_2))}\})\ a \in C'\ a$$

10. $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\mathsf{Elim}[\kappa, u](\tau)\{\vec{\kappa'}\}) = G(\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa))$

    where $\kappa = \mathsf{Ind}(j : \mathsf{Kind})\{\vec{\kappa}\}$ is well constructed of schema Kind under $\mathcal{K}$ and $\mathcal{U}$ and $G(C) \in \rho(u)_{\mathcal{K}}$ is defined for all $C \in dom(<_\kappa)$ as follows ($<_\kappa$ is the order induced by the inductive definition κ):

    - *If $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau)$ has a normal form $b = \mathsf{Co}(i)\ \vec{a}$ such that $b \in C$*

$$G(C) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U},\alpha_1:G(\mathsf{pred}_{(b)})}(\Phi_{j,I,\alpha_1}(\kappa_i, \kappa'_i))\ (\vec{a})$$

- $\mathsf{Can}(u)_{\mathcal{K}}$ *otherwise*

Lemma C.2.46 *The function F in Definition C.2.45.9 is monotonic.*

Proof We must prove that if $C_1 <_{\mathsf{Kind}} C_2$, then

$$\mathcal{C}^{\mathcal{K}}_{\mathcal{U},j:C_2,A':C',B':\mathsf{Co}(i)}(\zeta_{j,I}(\kappa_i, A', B')) <_{\mathsf{Kind}} \mathcal{C}^{\mathcal{K}}_{\mathcal{U},j:C_1,A':C',B':\mathsf{Co}(i)}(\zeta_{j,I}(\kappa_i, A', B'))$$

The proof is by induction on the fact that κ_i is the kind of a constructor.

- If $\kappa_i = j$, then both sides reduce to $C'\ \mathsf{Co}(i)$.
- If $\kappa_i = \Pi X : A_1.\, A_2$, then it follows directly from the induction hypothesis and because j does not occur in A_1.
- If $\kappa_i = \Pi \vec{X} : \vec{A}.\, j \to A_2$, then

  $$\zeta_{j,I}(\kappa_i, A', B') = \Pi Z : (\Pi \vec{X} : \vec{A}.\, j).\, \Pi \vec{X'} : \vec{A}.\, (A'\ (Z\ \vec{X'})) \to \zeta_{j,I}(A_2, A', B'\ Z)$$

  Suppose $\mathcal{U}' = \mathcal{U}, j : C'', A' : C', B' : \mathsf{Co}(i)$ where C'' is either C_1 or C_2. The required set is then

  $$\begin{gathered} a \in \Lambda, \text{ such that } \forall a_1 \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}'}(\Pi \vec{X} : \vec{A}.\, j), \\ \forall a_2 \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}', Z:a_1}(\Pi \vec{X'} : \vec{A}.\, A'\ (Z\ \vec{X'})) \\ a\ a_1\ a_2 \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}', Z:a_1}(\zeta_{j,I}(A_2, A', B'\ Z)) \end{gathered}$$

  The set of a_1 and a_2 is larger for the LHS. By the induction hypothesis, the result $a\ a_1\ a_2$ must occur in a smaller set for the LHS. The required result follows from this.

□

Remark C.2.47 *The previous lemma ensures that the interpretation of an inductive type sets up a well defined order. This ensures that the interpretation of large elimination (Definition C.2.45.10) is well formed.*

We get a bunch of substitution lemmas. The proof for each of these is similar and follows directly by induction over the structure of κ. We state them below:

Proposition C.2.48 *Let κ be a well constructed kind of schema u under $\mathcal{K}$ and $\mathcal{U}$. Let α be a type variable, and τ a type. We have that*

$$\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}([\tau/\alpha]\kappa) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U},\alpha:\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau)}(\kappa)$$

Proposition C.2.49 *Let κ be a well constructed kind of schema u under $\mathcal{K}$ and $\mathcal{U}$. Let j be a kind variable and κ_1 a kind such that κ_1 is well constructed under $\mathcal{K}$ and $\mathcal{U}$ of the same schema as $\mathcal{U}(j)$. We have that*

$$\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}([\kappa_1/j]\kappa) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U},j:\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_1)}(\kappa)$$

Proposition C.2.50 *Let κ be a well constructed kind of schema u under $\mathcal{K}$ and $\mathcal{U}$. Let z be a schema variable, and u_1 be a schema such that $\mathcal{K}(u_1)$ is an arity. We have that*

$$\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}([u_1/z]\kappa) = \mathcal{C}^{\mathcal{K},z:\mathcal{K}(u_1)}_{\mathcal{U}}(\kappa)$$

C.2.7 Candidate interpretation of kinds

Definition C.2.51 *We say that $\mathcal{U}$ and $\mathcal{U}'$ are equivalent interpretations if for all j, we have that $\mathcal{U}(j) \simeq \mathcal{U}'(j)$ and for all α we have that $\mathcal{U}(\alpha) =_{\beta\eta\iota} \mathcal{U}'(\alpha)$.*

Lemma C.2.52 *Let u be a schema, $\mathcal{K}$ be a mapping, and $\mathcal{U}$ and $\mathcal{U}'$ be two equivalent interpretations. Suppose κ is well constructed of schema $\mathcal{K}(u)$ under $\mathcal{K}$ and both $\mathcal{U}$ and $\mathcal{U}'$. Then $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa) \simeq_{\mathcal{K}(u)} \mathcal{C}^{\mathcal{K}}_{\mathcal{U}'}(\kappa)$.*

Proof The proof is by induction over the structure of κ. Most of the cases follow directly from the induction hypothesis.

- $\kappa = \mathsf{Elim}[\kappa', u](\tau)\{\vec{\kappa'}\}$. Here $\kappa' = \mathsf{Ind}(j : \mathsf{Kind})\{\vec{\kappa}\}$. First note that $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa') = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}'}(\kappa')$. Therefore, the function F whose lfp generates the inductive definition is the same. Moreover, $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau) =_{\beta\eta\iota} \mathcal{C}^{\mathcal{K}}_{\mathcal{U}'}(\tau)$. Since the set of pure terms is confluent, $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau)$ and $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}'}(\tau)$ have the same normal form. We can now do induction on the structure of κ_i to prove that

$$\mathcal{C}^{\mathcal{K}}_{\mathcal{U},\alpha_1:G(\mathsf{pred}(b))}(\Phi_{j,I,\alpha_1}(\kappa_i, \kappa'_i)) \simeq \mathcal{C}^{\mathcal{K}}_{\mathcal{U}',\alpha_1:G(\mathsf{pred}(b))}(\Phi_{j,I,\alpha_1}(\kappa_i, \kappa'_i))$$

□

Lemma C.2.53 *Let $\mathcal{K}$ be a mapping, $\mathcal{U}$ a candidate interpretation, κ be a kind and u be a schema such that κ is a well constructed kind of schema $\mathcal{K}(u)$. Then $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa) \in \mathcal{CR}(u)_{\mathcal{K}}$.*

Proof The proof is by induction over the structure of κ. Most of the cases follow in a direct way.

- $\kappa = \mathsf{Ind}(j : \mathsf{Kind})\{\vec{\kappa}\}$. We will use lemma C.2.38 to prove this. For $\mathcal{S} \in \mathcal{CR}_0(\mathsf{Kind})$, the first condition is satisfied by lemma C.2.30.

  – Suppose $\mathcal{S} = \bot$. If none of the branches is recursive then the function F is a constant function and the proof is similar to the non-bottom case. Suppose the ith branch is recursive. Then it is easy to see that the b_i defined as:

  $$b_i \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U},j:\bot,A':C,B':\mathsf{Co}(i)}(\zeta_{j,I}(\kappa_i, A', B'))$$

  includes the set of all terms, including non-normalizing ones. Therefore, there are no terms a that would satisfy the condition that:

  $$(\mathsf{match}\,\alpha.\{\overrightarrow{\mathcal{C}^{\mathcal{K}}_{\mathcal{U},a_i:b_i}(\Upsilon(\kappa_i, a_i, \lambda\alpha_2.\alpha\,\alpha_2))}\})\ a \in C\ a$$

  This implies that $F(\bot) = \bot$ and we know that $\bot \in \mathcal{CR}_0(\mathsf{Kind})$.

  – Consider any other $\mathcal{S}$. We will show that $F(\mathcal{S})$ satisfies the conditions in Definition C.2.10 and hence belongs to $\mathcal{CR}_0(\mathsf{Kind})$. $F(\mathcal{S})$ is defined as the union of $min(\mathsf{Kind})$ with the set of pure terms a such that

  $$(\mathsf{match}\,\alpha.\{\overrightarrow{\mathcal{C}^{\mathcal{K}}_{\mathcal{U},a_i:b_i}(\Upsilon(\kappa_i, a_i, \lambda\alpha_2.\alpha\,\alpha_2))}\})\ a \in C\ a$$

  Since C is a candidate, the terms a must be strongly normalizing.

  To see that the set is closed under reduction, suppose $a \rhd a'$. Since C is a candidate we have that $(\mathsf{match}\,\alpha.\{\ldots\})\ a' \in C\ a$. Moreover, we have that $C\ a = C\ a'$. Therefore, a' is also in the generated set.

  Suppose a is a neutral term and for all a' such that $a \rhd a'$, we have that a' belongs to this set. We have to prove that a belongs to this set. This implies that we must prove:

  $$(\mathsf{match}\,\alpha.\{\overrightarrow{\mathcal{C}^{\mathcal{K}}_{\mathcal{U},a_i:b_i}(\Upsilon(\kappa_i, a_i, \lambda\alpha_2.\alpha\,\alpha_2))}\})\ a \in C\ a$$

Since a is a neutral term, the above term does not have a redex at the head. From the induction hypothesis, we get that $\mathcal{C}^{\mathcal{K}}_{\mathcal{U},j:\mathcal{S},A':C,B':\mathsf{Co}(i)}(\zeta_{j,I}(\kappa_i, A', B'))$ is a candidate and therefore closed under reduction. Moreover, the b_i are strongly normalizing. We can now consider all possible redices and prove by induction over $\nu(b_i)$ that the above condition is satisfied.

- $\kappa = \mathsf{Elim}[\kappa', u](\tau)\{\vec{\kappa'}\}$ where $\kappa' = \mathsf{Ind}(j : \mathsf{Kind})\{\vec{\kappa}\}$. First note that $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa')$ is a candidate by induction and gives rise to a well founded order on $\mathcal{CR}_0(\mathsf{Kind})$. We will do induction on this order. Suppose $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa) = G(\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa'))$. We will show that for all sets $\mathcal{S}$ belonging to the order generated by κ', and for all pure terms b, we have that $G(\mathcal{S}) \in \mathcal{CR}(u)_{\mathcal{K}}$. For the non-recursive case, the proof is immediate. For the recursive case, consider $\mathcal{C}^{\mathcal{K}}_{\mathcal{U},\alpha_1:G(\mathsf{pred}(b))}(\Phi_{j,\kappa',\alpha_1}(\kappa_i, \kappa'_i))$. Note that $\mathsf{pred}(b)$ belongs to the same order. The required result follows now by doing induction over the structure of κ_i and applying the induction hypothesis to $G(\mathsf{pred}(b))$.

□

Definition C.2.54 *Suppose Δ is a context and $\mathcal{K}$ and $\mathcal{U}$ are a mapping and an interpretation. We say that $\mathcal{K}$ and $\mathcal{U}$ are adapted to Δ if:*

- $\forall z \in \Delta$, *we have that* $\mathcal{K}(z)$ *is an arity and* $\cdot \vdash \mathcal{K}(z) : \mathsf{Kscm}$.
- $\forall j \in \Delta$, *we have that* $\mathcal{U}(j) \in \mathcal{CR}(\Delta(j))_{\mathcal{K}}$.
- $\forall \alpha \in \Delta$, *we have that* $\mathcal{U}(\alpha) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\Delta(\alpha))$.

C.2.8 Interpretation of abstractions

We get a bunch of lemmas that state that an abstraction at the type level belongs to the corresponding kind. The proof of each of these lemmas is straightforward and follows in a similar way. We will show the proof for only one of the lemmas.

Lemma C.2.55 *Let $\Delta \vdash \lambda\alpha : \kappa.\, \tau : \Pi\alpha : \kappa.\, \kappa_1$ be a judgment and $\mathcal{K}$ and $\mathcal{U}$ be a mapping and a candidate interpretation adapted to Δ. We have $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\lambda\alpha : \kappa.\, \tau) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\Pi\alpha : \kappa.\, \kappa_1)$ if and only if for all pure terms $a \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa)$, we have that $\mathcal{C}^{\mathcal{K}}_{\mathcal{U},\alpha:a}(\tau) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U},\alpha:a}(\kappa_1)$.*

Lemma C.2.56 *Let $\Delta \vdash \lambda j : u.\, \tau : \Pi j : u.\, \kappa$ be a judgment and $\mathcal{K}$ and $\mathcal{U}$ be a mapping and a candidate interpretation adapted to Δ. We have $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\lambda j : u.\, \tau) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\Pi j : u.\, \kappa)$ if and only if for all reducibility candidates $C \in \mathcal{CR}(u)_{\mathcal{K}}$ we have that $\mathcal{C}^{\mathcal{K}}_{\mathcal{U},j:C}(\tau) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U},j:C}(\kappa)$.*

Lemma C.2.57 *Let $\Delta \vdash \lambda z : \mathsf{Kscm}.\, \tau : \Pi z : \mathsf{Kscm}.\, \kappa$ be a judgment and $\mathcal{K}$ and $\mathcal{U}$ be a mapping and a candidate interpretation adapted to Δ. We have $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\lambda z : \mathsf{Kscm}.\, \tau) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\Pi z : \mathsf{Kscm}.\, \kappa)$ if and only if for all $u \in \mathsf{arity}(u', \mathsf{Kind})$ we have that $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau) \in \mathcal{C}^{\mathcal{K},z:u}_{\mathcal{U}}(\kappa)$.*

Proof By definition $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\lambda z : \mathsf{Kscm}.\, \tau) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau)$. Similarly $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\Pi z : \mathsf{Kscm}.\, \kappa) = \cap_{u_1 \in \mathsf{arity}(u,\mathsf{Kind})} \mathcal{C}^{\mathcal{K},z:u_1}_{\mathcal{U}}(\kappa)$. The *if* part follows directly from the definition.

For the *only if*, suppose that $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau) \in \mathcal{C}^{\mathcal{K},z:u}_{\mathcal{U}}(\kappa)$ for all arities u. This implies that $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau) \in \cap_{u_1 \in \mathsf{arity}(u,\mathsf{Kind})} \mathcal{C}^{\mathcal{K},z:u_1}_{\mathcal{U}}(\kappa)$. This implies that $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\Pi z : \mathsf{Kscm}.\, \kappa)$. □

C.2.9 Interpretation of weak elimination

For this section $\kappa = \mathsf{Ind}(j : \mathsf{Kind})\{\vec{\kappa}\}$. Suppose also that $C \in \mathcal{CR}(\kappa \to \mathsf{Kind})_{\mathcal{K}}$ and $\tau_i \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U},A':C,B':\mathsf{Co}_{(i)}}(\zeta_{j,I}(\kappa_i, A', B'))$.

Lemma C.2.58 *Suppose $a \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa)$. We have then*

$$(\mathsf{match}\, \alpha.\{\overrightarrow{\Upsilon(\kappa_i, \tau_i, \lambda\alpha_2.\alpha\, \alpha_2)}\})\ a \in C\ a$$

Proof Follows immediately from the definition of $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa)$. □

Lemma C.2.59 *Let $\Delta \vdash \mathsf{Elim}[\kappa, \kappa_1](\tau)\{\vec{\tau'}\} : \kappa_1$ be a derivable judgment where κ_1 is a kind. Suppose $\mathcal{K}$ is a mapping and $\mathcal{U}$ is a candidate interpretation adapted to Δ. If $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa)$ and $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau'_i) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\zeta_{j,I}(\kappa_i, \kappa_1, \mathsf{Ctor}\,(i, \kappa)))$, then we have*

$$\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\mathsf{Elim}[\kappa, \kappa_1](\tau)\{\vec{\tau'}\}) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_1)$$

Proof Follows now from the previous lemma. □

C.2.10 Interpretation of constructors

For this section, suppose $I = \kappa = \mathsf{Ind}(j : \mathsf{Kind})\{\vec{\kappa}\}$. Also, suppose $C \in \mathcal{CR}(I \to \mathsf{Kind})_{\mathcal{K}}$.

Lemma C.2.60 *For all i, $\mathsf{Co}(i) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_i)$.*

Proof We know that κ_i is of the form $\Pi \vec{X} : \vec{A}.\, j$. Suppose $\vec{B} \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U},j:\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(I)}(\vec{X} : \vec{A})$. Then we need to prove that $\mathsf{Co}(i)\ \vec{B} \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(I)$. This means that we need to prove that

$$(\mathsf{match}\, \alpha.\{\overrightarrow{\Upsilon(\kappa_i, a_i, \lambda\alpha_2.\alpha\, \alpha_2)}\})\ (\mathsf{Co}(i)\ \vec{B}) \in C\ (\mathsf{Co}(i)\ \vec{B})$$

where a_i belongs to the appropriate candidate. This implies that we need to prove that

$$\Upsilon(\kappa_i, a_i, \lambda\alpha_2.\mathsf{match}\, \alpha.\{\ldots\}\, \alpha_2)\ \vec{B} \in C\ (\mathsf{Co}(i)\ \vec{B})$$

This follows directly by an induction over the structure of κ_i. □

C.2.11 Invariance under β reduction

In this section, we show that the interpretation of kinds remains invariant under β reduction.

Lemma C.2.61 *Let κ be a well constructed kind of schema u under a mapping $\mathcal{K}$ and candidate interpretation $\mathcal{U}$. If $\kappa \rhd_\beta \kappa'$, then κ' is well constructed of schema u under $\mathcal{K}$ and $\mathcal{U}$, and $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa')$.*

Proof The proof is by induction over the structure of κ. Most of the cases follow directly from the induction hypothesis. We will only consider β reductions at the head.

- $\kappa = (\lambda\alpha : \kappa_1.\, \kappa_2)\ \tau$. By definition,

  $$\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}((\lambda\alpha : \kappa_1.\, \kappa_2)\ \tau) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\lambda\alpha : \kappa_1.\, \kappa_2)\ \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau)$$

  Again by definition this is equal to $\mathcal{C}^{\mathcal{K}}_{\mathcal{U},\alpha:\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau)}(\kappa_2)$. By proposition C.2.48 this is equal to $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}([\tau/\alpha]\kappa_2)$

- $\kappa = (\lambda j : u_1.\, \kappa_1)\ \kappa_2$. By definition,

  $$\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}((\lambda j : u_1.\, \kappa_1)\ \kappa_2) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\lambda j : u_1.\, \kappa_1)\ \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_2)$$

By lemma C.2.53 we have that $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_2) \in \mathcal{CR}(u_1)_{\mathcal{K}}$. Therefore, we get that

$$\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}((\lambda j\!:\!u_1.\,\kappa_1)\ \kappa_2) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U},j:\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_2)}(\kappa_1)$$

By proposition C.2.49 this is equal to $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}([\kappa_2/j]\kappa_1)$.

□

C.2.12 Invariance under η reduction

In this section, we show that the interpretation remains the same under η reduction. The unmarked terms $\|\,\kappa\,\|$ are defined in Section C.1.1.

Lemma C.2.62 *Let κ be a well constructed kind of schema u under a mapping $\mathcal{K}$ and candidate interpretation $\mathcal{U}$. If $\kappa \rhd_\eta \kappa'$, then κ' is well constructed of schema u under $\mathcal{K}$ and $\mathcal{U}$, and $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa')$.*

Proof The proof is again by induction over the structure of κ. We will consider only the cases where the reduction occurs at the head.

- $\kappa = \lambda\alpha\!:\!\kappa_1.\,(\kappa_2\ \alpha)$. By definition $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa)$ is equal to:

  $$a \in \Lambda \longmapsto \mathcal{C}^{\mathcal{K}}_{\mathcal{U},\alpha:a}(\kappa_2)\ \mathcal{C}^{\mathcal{K}}_{\mathcal{U},\alpha:a}(\alpha)$$

  Since α does not occur free in κ_2, this is equivalent to

  $$a \in \Lambda \longmapsto \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_2)\ a$$

  Since a does not occur free now in $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_2)$, we get that this is equivalent to $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_2)$. Note from Definition C.2.11 that the domain of $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_2)$ is Λ.

- $\kappa = \lambda j\!:\!u_1.\,(\kappa_2\ j)$. By definition $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa)$ is equal to:

  $$C \in \mathcal{CR}(u_1)_{\mathcal{K}} \longmapsto \mathcal{C}^{\mathcal{K}}_{\mathcal{U},j:C}(\kappa_2)\ \mathcal{C}^{\mathcal{K}}_{\mathcal{U},j:C}(j)$$

Since j does not occur free in κ_2, this is equivalent to

$$C \in \mathcal{CR}(u_1)_{\mathcal{K}} \longmapsto \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_2)\ C$$

Since C does not occur free now in $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_2)$, we get that this is equivalent to $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_2)$. Note from Definition C.2.11 that the domain of $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_2)$ is $\mathcal{CR}(u_1)_{\mathcal{K}}$.

□

Lemma C.2.63 *For all well constructed kinds κ of schema u under $\mathcal{K}$ and $\mathcal{U}$, we have $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\|\kappa\|)$.*

Proof Follows from the fact that $\kappa =_{\beta\eta} \|\kappa\|$. □

C.2.13 Invariance under ι reduction

In this section we essentially show that interpretation remains the same under large elimination.

Lemma C.2.64 *Let $\mathsf{Elim}[\kappa, u](\tau)\{\vec{\kappa'}\}$ be well constructed of schema $\mathcal{K}(u)$ under $\mathcal{K}$ and $\mathcal{U}$. Suppose $\kappa = \mathsf{Ind}(j : \mathsf{Kind})\{\vec{\kappa}\}$. Suppose G is the function used for the interpretation of the large elimination. If $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa)$, then for all $C \in \mathcal{CR}_0(\mathsf{Kind})$ with $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau) \in C$, we have that $G(\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa)) = G(C)$.*

Proof The proof is immediate. □

Lemma C.2.65 *Suppose $I = \kappa = \mathsf{Ind}(j : \mathsf{Kind})\{\vec{\kappa}\}$. Suppose the constructors of I are all small. Suppose the mth constructor of I has the form $\Pi\vec{Y} : \vec{B}.\ j$ and we have a sequence of terms $\vec{b}$ such that $\mathsf{Co}(m)\ \vec{b} \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(I)$. Then we have that $b_i \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U},\forall k<i.Y_k:b_k,j:\mathsf{pred}(\mathsf{Co}(m)\ \vec{b})}(B_i)$.*

Proof We can have two cases.

- $\mathsf{pred}(\mathsf{Co}(m)\ \vec{b}) \neq \bot$

  This implies that $\mathsf{pred}(\mathsf{Co}(m)\ \vec{b}) \in \mathcal{CR}_0(\mathsf{Kind})$. Suppose $\mathcal{S} = \mathcal{C}^{\mathcal{K}}_{\mathcal{U},\forall k<i.Y_k:b_k,j:\mathsf{pred}(\mathsf{Co}(m)\ \vec{b})}(B_i)$. Then we have that $\mathcal{S}$ is a candidate of schema Kind. Suppose also that C' belongs to $\mathcal{CR}(I \to \mathsf{Kind})_{\mathcal{K}}$ and maps elements in the domain of $I \to \mathsf{Kind}$ to $\mathcal{S}$. Then for all indices i', we have that $\mathcal{C}^{\mathcal{K}}_{\mathcal{U},j:\mathsf{pred}(\mathsf{Co}(m)\ \vec{b}),A':C'}(\zeta_{j,I}(\kappa_{i'}, A', \mathsf{Ctor}\ (i', I)))$ is a reducibility candidate of Kind.

To prove the lemma we need to show that if for all indices i

$$\tau_i \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U},j:\mathsf{pred}(\mathsf{Co}(m)\ \vec{b}),A':C'}(\zeta_{j,I}(\kappa_i, A', \mathsf{Ctor}\ (i, I)))$$

then we have that $\Phi_{j,I,B'}(\kappa_m, \tau_m)$ can reduce to b_i by a head reduction. To have this, for the indices $i \neq m$ choose τ_i as some variable. For τ_m choose the term that returns the ith argument of the constructor.

- $\mathsf{pred}(\mathsf{Co}(m)\ \vec{b}) = \bot$ We can show that the constructors now are not recursive. Hence j does not occur free in any of the B_is. The proof for the previous case can be reused here.

□

Lemma C.2.66 *Let $\Delta \vdash \mathsf{Elim}[\kappa, u](\tau)\{\vec{\kappa'}\} : u$ be a derivable judgment. Let $\mathcal{K}$ be a mapping and $\mathcal{U}$ be an interpretation adapted to Δ. Suppose $I = \kappa = \mathsf{Ind}(j : \mathsf{Kind})\{\vec{\kappa}\}$. Suppose $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa)$ and $\tau \rhd^* \mathsf{Ctor}\ (i, \kappa)\ \vec{A}$. Also suppose $B' = \lambda\alpha : I.\, \mathsf{Elim}[\kappa, u](\alpha)\{\vec{\kappa'}\}$. We then have that $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\mathsf{Elim}[\kappa, u](\tau)\{\vec{\kappa'}\}) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\Phi_{j,I,B'}(\kappa_i, \kappa'_i)\ (\vec{A}))$.*

Proof Let G be the function used for interpreting large elimination. Suppose $\mathsf{Co}(i)\ \vec{a}$ is the normal form of $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau)$. Then given the assumptions we have that:

$$\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\mathsf{Elim}[\kappa, u](\tau)\{\vec{\kappa'}\}) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U},B':G(\mathsf{pred}(\mathsf{Co}(i)\ \vec{a}))}(\Phi_{j,I,B'}(\kappa_i, \kappa'_i))\ (\vec{a})$$

We therefore have to prove that

$$\mathcal{C}^{\mathcal{K}}_{\mathcal{U},B':G(\mathsf{pred}(\mathsf{Co}(i)\ \vec{a}))}(\Phi_{j,I,B'}(\kappa_i, \kappa'_i))\ (\vec{a}) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\Phi_{j,I,B'}(\kappa_i, \kappa'_i)\ (\vec{A}))$$

- $\kappa_i = j$ it follows directly.
- $\kappa_i = \Pi\alpha : \kappa_1.\, \kappa_2$ We have to prove that

  $$\mathcal{C}^{\mathcal{K}}_{\mathcal{U},B':G(\mathsf{pred}(\mathsf{Co}(i)\ \vec{a})),\alpha:a_1}(\Phi_{j,I,B'}(\kappa_2, \kappa'_i\ \alpha))\ (a_{2..n}) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U},\alpha:a_1}(\Phi_{j,I,B'}(\kappa_2, \kappa'_i\ \alpha)\ (A_{2..n}))$$

  Applying the induction hypothesis leads to the result.

- $\kappa_i = \Pi\vec{\alpha}{:}\vec{\kappa}.\, j \to \kappa_2$ The LHS becomes

$$\mathcal{C}^{\mathcal{K}}_{\mathcal{U}'}(\Phi_{j,I,B'}(\kappa_2, \kappa_i'\alpha(\lambda\vec{Y}{:}\vec{\kappa}.\, B'(\alpha\vec{Y})))) \ (a_{2..n})$$

$$\text{where } \mathcal{U}' = \mathcal{U}, B' : G(\mathsf{pred}(\mathsf{Co}(i)\ \vec{a})), \alpha : a_1$$

By lemma C.2.65, a_1 belongs to $\mathcal{C}^{\mathcal{K}}_{\mathcal{U},j:\mathsf{pred}(\mathsf{Co}(i)\ \vec{a})}(\Pi\vec{\alpha}{:}\vec{\kappa}.\, j)$. This implies that $a_1\ \vec{Y} \in \mathsf{pred}(\mathsf{Co}(i)\ \vec{a})$. Moreover, by lemma C.2.64 $G(\mathsf{pred}(\mathsf{Co}(i)\ \vec{a}))(a_1\ \vec{Y})$ is equal to $G(\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa))(a_1\ \vec{Y})$ and which is in turn equal to $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\mathsf{Elim}[\kappa, u](A_1\ \vec{Y})\{\vec{\kappa'}\})$. The required result follows directly from here by performing one head reduction on the RHS and applying the induction hypothesis.

□

C.2.14 Kinds invariant on their domain

Definition C.2.67 *Let $\Delta \vdash \kappa : u$ be a derivable judgment and $\mathcal{K}$ and $\mathcal{U}$ be a mapping and an interpretation adapted to Δ. We say $(\kappa, u, \Delta, \mathcal{K}, \mathcal{U})$ is invariant if:*

- $u = \mathsf{Kind}$ *and for all κ' such that $\kappa \rhd^* \kappa'$, we have that $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa')$;*
- $u = \Pi\alpha{:}\kappa_1.\, u_1$ *then for all derivable judgments $\Delta \vdash \tau : \kappa_1$ and $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_1)$, we have that $(\kappa\ \tau, [\tau/\alpha]u_1, \Delta, \mathcal{K}, \mathcal{U})$ is invariant;*
- $u = \Pi j{:}u_1.\, u_2$ *then for all derivable judgments $\Delta \vdash \kappa_1 : u_1$, we have that $(\kappa\ \kappa_1, [\kappa_1/j]u_2, \Delta, \mathcal{K}, \mathcal{U})$ is invariant;*
- $u = z$ *and we have that $(\mathcal{K}(\kappa), \mathcal{K}(u), \mathcal{K}(\Delta), \mathcal{K}, \mathcal{U})$ is invariant.*

Lemma C.2.68 *Let $\Delta \vdash \kappa_1 : \mathsf{Kind}$ and $\Delta \vdash \kappa_2 : \mathsf{Kind}$ be two derivable judgments and $\mathcal{K}$ and $\mathcal{U}$ be a mapping and an interpretation adapted to Δ. If $(\kappa_1, \mathsf{Kind}, \Delta, \mathcal{K}, \mathcal{U})$ and $(\kappa_2, \mathsf{Kind}, \Delta, \mathcal{K}, \mathcal{U})$ are invariant and $\kappa_1 =_{\beta\eta\iota} \kappa_2$, then $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_1) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_2)$.*

Proof We know that there exists a B such that $\| \kappa_1 \| \rhd^* B$ and $\| \kappa_2 \| \rhd^* B$. This implies that there exists a κ_1' and a κ_2' (lemma C.1.4 and C.1.14) such that $\kappa_1 \rhd_{\beta\iota} \kappa_1'$ and $\| \kappa_1' \| \rhd^*_\eta B$. Similarly, $\kappa_2 \rhd_{\beta\iota} \kappa_2'$ and $\| \kappa_2' \| \rhd^*_\eta B$. From here we get that

$$\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_1) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_1') = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(B) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_2') = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_2)$$

□

Proposition C.2.69 *If* $([\tau/\alpha]\kappa, u, \Delta, \mathcal{K}, \mathcal{U})$ *is invariant, and also* $\Delta \vdash (\lambda\alpha : \kappa_1 . \kappa)\ \tau : u$*, then* $((\lambda\alpha : \kappa_1 . \kappa)\ \tau, u, \Delta, \mathcal{K}, \mathcal{U})$ *is invariant.*

C.2.15 Interpretation of large elimination

Lemma C.2.70 *Let* $\Delta \vdash \mathsf{Elim}[\kappa, u](\tau)\{\vec{\kappa'}\} : u$ *be a judgment. Suppose* $I = \kappa = \mathsf{Ind}(j : \mathsf{Kind})\{\vec{\kappa}\}$*. Let* $\mathcal{K}$ *and* $\mathcal{U}$ *be a mapping and an interpretation adapted to* Δ*. Suppose*

1. $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa)$.
2. *for all* i, $(\kappa'_i, \Psi_{j,I}(\kappa_i, u), \Delta, \mathcal{K}, \mathcal{U})$ *is invariant.*

Then we have that $(\mathsf{Elim}[\kappa, u](\tau)\{\vec{\kappa'}\}, u, \Delta, \mathcal{K}, \mathcal{U})$ *is invariant.*

Proof Suppose $\kappa_1 = \mathsf{Elim}[\kappa, u](\tau)\{\vec{\kappa'}\}$. Suppose we are given a sequence of terms $\vec{A}$ of the proper type so that $\kappa_1\ \vec{A}$ is in Kind. To show the invariance, we have to show that if $\kappa_1\ \vec{A} \rhd^* \kappa_2$, then $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_1\ \vec{A}) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_2)$. We will reason by induction on $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau)$ over the order defined by I.

- If the term $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau)$ can not be reduced to a term of the form $\mathsf{Co}(i)\vec{a}$, then it is minimal with respect to the order defined by I. Then κ_2 is necessarily of the form $\mathsf{Elim}[\kappa', u'](\tau')\{\vec{\kappa''}\}\ \vec{A'}$ and we have that the interpretation of both $\kappa_1\ \vec{A}$ and κ_2 is $\mathsf{Can}_0(\mathsf{Kind})$.
- Suppose the term $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau)$ can be reduced to a term of the form $\mathsf{Co}(i)\vec{a}$, but τ is not reduced to a term of the form $\mathsf{Ctor}\,(i, I)\vec{C}$. Then κ_2 is again of the form $\mathsf{Elim}[\kappa''', u'](\tau')\{\vec{\kappa''}\}\ \vec{A'}$. By definition, we have that

$$B_1 = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_1\ \vec{A}) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U},\vec{\alpha}:\vec{a}}(\Phi_{j,I,B'}(\kappa_i, \kappa'_i)\ (\vec{\alpha})\vec{A})$$
$$B_2 = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_2) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U},\vec{\alpha}:\vec{a}}(\Phi_{j,I',B''}(\vec{\kappa'''_i}, \vec{\kappa''_i})\ (\vec{\alpha})\vec{A'})$$

  where $B' = \lambda Y : I . \mathsf{Elim}[\kappa, u](\tau)\{\vec{\kappa'}\}$, and $B'' = \lambda Y : I' . \mathsf{Elim}[\kappa''', u'](\tau')\{\vec{\kappa''}\}$. It is evident that B_2 is a reduct of B_1, and therefore we need to prove that $(B_1, \mathsf{Kind}, \Delta, \mathcal{K}, \mathcal{U})$ is invariant.

  This follows by an induction over the structure of κ_i and by using the condition 2. The non-recursive cases follow directly. For the recursive case, we use lemma C.2.65 to show that B' is applied to a smaller argument with respect to the order defined by I.

- We are left with the case when τ reduces to a term of the form $\mathsf{Ctor}\,(i,I)\vec{C}$. In going from $\kappa_1\ \vec{A}$ to κ_2, we will now have a ι reduction. The sequence of reductions is now

$$\begin{aligned}\kappa_1\ \vec{A} &\rhd^* \mathsf{Elim}[\kappa,u](\mathsf{Ctor}\,(i,I)\vec{A})\{\vec{\kappa'}\}\\ &\rhd_\iota (\Phi_{j,I,\kappa_i'}(\kappa_i,B')\ (\vec{B}))\vec{A}\\ &\rhd^* \kappa_2\end{aligned}$$

  The first reduction does not change the interpretation since we are reducing only a type. By lemma C.2.66, the second does not change the interpretation. Finally, as above, we can prove that the result of the ι reduction is invariant over Kind.

□

C.2.16 Instantiation of contexts

Definition C.2.71 *Let Δ be a well formed context. Let Θ be a context and ϕ be a mapping from variables to terms such that $\forall X \notin \Delta, \phi(X) = X$.*

We say that (Θ,ϕ) is an instantiation of Δ if for all variables $X \in \Delta$, we have that $\Theta \vdash \phi(X) : \phi(\Delta(X))$.

Lemma C.2.72 *Let $\Delta \vdash A : B$ be a derivable judgment and (Θ,ϕ) an instantiation of Δ. Then $\Theta \vdash \phi(A) : \phi(B)$.*

Proof By induction over the structure of A. □

Definition C.2.73 (Adapted instantiation) *We say that an instantiation (Θ,ϕ) is adapted to a context Δ if:*

- *for all $\alpha \in \Delta$, $\phi(\alpha) \in \mathcal{C}^{\emptyset}_{\mathsf{Can}_0(\Theta)}(\phi(\Delta(\alpha)))$;*
- *for all $j \in \Delta$, $(\phi(j), \phi(\Delta(j)), \Theta, \emptyset, \mathsf{Can}_0(\Theta))$ is invariant;*
- *for all $z \in \Delta$, $(\phi(z), \mathsf{Kscm}, \Theta, \emptyset, \mathsf{Can}_0(\Theta))$ is invariant and $\phi(z)$ is an arity.*

Definition C.2.74 *Suppose $\Delta \vdash \kappa : u$ is a derivable judgment. We say that all instantiations of (κ,u,Δ) are invariant if for all instantiations (Θ,ϕ) adapted to Δ and for all interpretations $\mathcal{U}$ adapted to Θ, we have that $(\phi(\kappa), \phi(u), \Theta, \emptyset, \mathcal{U})$ is invariant.*

C.2.17 Kind schema invariant on their domain

Definition C.2.75 *Let* $\Delta \vdash u : \mathsf{Kscm}$ *be a derivable judgment and* $\mathcal{K}$ *and* $\mathcal{U}$ *be a mapping and an interpretation adapted to* Δ*. We say that* $(u, \mathsf{Kscm}, \Delta, \mathcal{K}, \mathcal{U})$ *is invariant:*

- *if* $u = \mathsf{Kind}$*, then* $(u, \mathsf{Kscm}, \Delta, \mathcal{K}, \mathcal{U})$ *is invariant;*
- *if* $u = \Pi\alpha : \kappa_1 . u_1$*, then it is invariant if and only if* $(\kappa_1, \mathsf{Kind}, \Delta, \mathcal{K}, \mathcal{U})$ *is invariant and for all terms* τ *such that* $\Delta \vdash \tau : \kappa_1$ *is derivable and* $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_1)$*, we have that* $([\tau/\alpha]u_1, \mathsf{Kscm}, \Delta, \mathcal{K}, \mathcal{U})$ *is invariant;*
- *if* $u = \Pi j : u_1 . u_2$*, then it is invariant if and only if* $(u_1, \mathsf{Kscm}, \Delta, \mathcal{K}, \mathcal{U})$ *is invariant, and for all kinds* κ *such that* $\Delta \vdash \kappa : u_1$ *is derivable and* $(\kappa, u_1, \Delta, \mathcal{K}, \mathcal{U})$ *is invariant, we have that* $([\kappa/j]u_2, \mathsf{Kscm}, \Delta, \mathcal{K}, \mathcal{U})$ *is invariant;*
- *if* $u = z$*, then it is invariant* iff $(\mathcal{K}(z), \mathsf{Kscm}, \Delta, \mathcal{K}, \mathcal{U})$ *is invariant.*

Lemma C.2.76 *Let* $\Delta \vdash \kappa : u$ *and* $\Delta \vdash u' : \mathsf{Kscm}$ *be derivable judgments. Let* $\mathcal{K}$ *and* $\mathcal{U}$ *be a mapping and an interpretation adapted to* Δ*. Suppose* $u =_{\beta\eta\iota} u'$*, and* $(u, \mathsf{Kscm}, \Delta, \mathcal{K}, \mathcal{U})$ *and* $(u', \mathsf{Kscm}, \Delta, \mathcal{K}, \mathcal{U})$ *are invariant. If* $(\kappa, u, \Delta, \mathcal{K}, \mathcal{U})$ *is invariant, then* $(\kappa, u', \Delta, \mathcal{K}, \mathcal{U})$ *is also invariant.*

Proof The proof is by induction over the structure of u and u'.

- if $u = u' = \mathsf{Kind}$, then it is trivially true.
- if $u = u' = z$, then again it is trivially true.
- if $u = \Pi\alpha : \kappa_1 . u_1$ and $u' = \Pi\alpha : \kappa_2 . u_2$, then we have that $\kappa_1 =_{\beta\eta\iota} \kappa_2$ and $u_1 =_{\beta\eta\iota} u_2$. By assumption, we know that $(\kappa_1, \mathsf{Kind}, \Delta, \mathcal{K}, \mathcal{U})$ and $(\kappa_2, \mathsf{Kind}, \Delta, \mathcal{K}, \mathcal{U})$ are invariant. This means that $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_1) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa_2)$. Moreover, $\Delta \vdash \tau : \kappa_1$ is true *iff* $\Delta \vdash \tau : \kappa_2$ is true. Applying the induction hypothesis now leads to the required result.
- if $u = \Pi j : u_1 . u_2$ and $u' = \Pi j : u'_1 . u'_2$, the proof is similar to the previous case.

□

Definition C.2.77 *Suppose* $\Delta \vdash u : \mathsf{Kscm}$ *is a derivable judgment. We say that all instantiations of* $(u, \mathsf{Kscm}, \Delta)$ *are invariant if for all instantiations* (Θ, ϕ) *adapted to* Δ *and for all interpretations* $\mathcal{U}$ *adapted to* Θ*, we have that* $(\phi(u), \mathsf{Kscm}, \Theta, \emptyset, \mathcal{U})$ *is invariant.*

C.2.18 Strong normalization of pure terms

Theorem C.2.78 *Let $\Delta \vdash \tau : \kappa$ be a derivable judgment and $\mathcal{K}$ and $\mathcal{U}$ be a mapping and an interpretation adapted to Δ. Then $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa)$.*

Proof The proof is by induction over the length of the derivation. The induction hypothesis are as follows:

- if $\Delta \vdash \tau : \kappa$ and $\mathcal{K}$ and $\mathcal{U}$ be a mapping and an interpretation adapted to Δ, then $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\kappa)$;
- if $\Delta \vdash \kappa : u$, then all instantiations of (κ, u, Δ) are invariant;
- if $\Delta \vdash u : \mathsf{Kscm}$, then all instantiations of $(u, \mathsf{Kscm}, \Delta)$ are invariant;

type formation rules This paragraph deals with rules of the form $\Delta \vdash \tau : \kappa$.

- abstractions – Follows directly from the induction hypothesis and lemmas C.2.55 and C.2.56 and C.2.57.
- var – Follows because the interpretation $\mathcal{U}$ is adapted to the context Δ.
- weak elimination – Follows from lemma C.2.59.
- constructor – Follows from lemma C.2.60.
- weakening – Follows directly from the induction hypothesis since the mapping and interpretation remain adapted for a smaller context.
- conv – Follows from the recursion hypothesis and lemma C.2.68.
- app – All three cases of app are proved similarly. We will show only one case here.
  - $\Delta \vdash \tau[u'] : \kappa$. Then we know that $\Delta \vdash \tau : \Pi z : \mathsf{Kscm}.\, \kappa_1$ and $\Delta \vdash u' : \mathsf{Kscm}$ and $[u'/z]\kappa_1 = \kappa$. By the induction hypothesis

    $$\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau) \in \cap_{u_1 \in \mathsf{arity}(u,\mathsf{Kind})} \mathcal{C}^{\mathcal{K}, z:u_1}_{\mathcal{U}}(\kappa_1)$$

    Suppose $u'_1 = \mathcal{K}(u')$. Then we know that $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau) \in \mathcal{C}^{\mathcal{K}, z:u'_1}_{\mathcal{U}}(\kappa_1)$. By proposition C.2.50 we know that $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau) \in \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}([u'/z]\kappa_1)$. But $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau[u']) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau)$.

kind formation rules This paragraph deals with rules of the form $\Delta \vdash \kappa : u$.

- product – All the product formation rules are proved in the same way. We show only one case here.

  – Consider the following formation rule

  $$\frac{\Delta, z{:}\mathsf{Kscm} \vdash \kappa : \mathsf{Kind}}{\Delta \vdash \Pi z{:}\mathsf{Kscm}.\, \kappa : \mathsf{Kind}}$$

  We have to prove that for all instantiations (Θ, ϕ) we have that $(\Pi z : \mathsf{Kscm}.\, \phi(\kappa), \mathsf{Kind}, \Theta, \emptyset, \mathcal{U})$ is invariant. This implies that we must prove that if $\kappa \rhd \kappa'$, then $\mathcal{C}^{\emptyset}_{\mathcal{U}}(\Pi z : \mathsf{Kscm}.\, \phi(\kappa)) = \mathcal{C}^{\emptyset}_{\mathcal{U}}(\Pi z : \mathsf{Kscm}.\, \phi(\kappa'))$. By the induction hypothesis, for all instantiations $(\Theta, \phi; z{:}\mathsf{arity}(u, \mathsf{Kind}))$ we have that

  $$(\phi; z{:}\mathsf{arity}(u, \mathsf{Kind})(\kappa), \mathsf{Kind}, \Theta, \emptyset, \mathcal{U})$$

  is invariant. This implies that if $\kappa \rhd \kappa'$ then

  $$\mathcal{C}^{,z:\mathsf{arity}(u,\mathsf{Kind})}_{\mathcal{U}}(\kappa) = \mathcal{C}^{,z:\mathsf{arity}(u,\mathsf{Kind})}_{\mathcal{U}}(\kappa')$$

  The required result follows from here.

- var – follows since the instantiation is adapted.

- conv – follows from lemma C.2.76.

- application – Both of the applications are proved similarly and follow directly from the induction hypothesis. We will show only one case here.

  – If $\Delta \vdash \kappa_1\ \kappa_2 : [\kappa_2/j]u$, then given Θ, ϕ, and $\mathcal{U}$, we must prove that $(\phi(\kappa_1\ \kappa_2), \phi([\kappa_2/j]u), \Theta, \emptyset, \mathcal{U})$ is invariant. But by the induction hypothesis we know that $(\phi(\kappa_1), \phi(\Pi j : u_1.\, u), \Theta, \emptyset, \mathcal{U})$ is invariant and $\Delta \vdash \kappa_2 : u_1$. By lemma C.2.72 $\Theta \vdash \phi(\kappa_2) : \phi(u_1)$. This leads to the required result.

- ind – Suppose $I = \mathsf{Ind}(j{:}\mathsf{Kind})\{\vec{\kappa}\}$. Note that $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(I)$ depends only on $\mathcal{C}^{\mathcal{K}}_{\mathcal{U},j:\mathcal{S},A':C,B':\mathsf{Co}_{(i)}}(\zeta_{j,I}(\kappa_i, A', B'))$ where $\mathcal{S} \in \rho_0(\mathsf{Kind})$ and $C \in \mathcal{CR}(I \to \mathsf{Kind})_{\mathcal{K}}$. By induction on the structure of κ_i, we can show that this is invariant. This implies that if $\kappa_i \rhd \kappa_i'$

then the interpretation remains the same. If $I \rhd I'$, then for some i, $\kappa_i \rhd \kappa_i'$. From here we can deduce that if $I \rhd I'$, then $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(I) = \mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(I')$.

- large elim – Follows from lemma C.2.70.
- abstraction – Both of the abstractions are proved similarly. So we will show only one of the cases.
  - $\Delta \vdash \lambda\alpha{:}\kappa_1.\,\kappa_2 : \Pi\alpha{:}\kappa_1.\,u$. We must prove that $(\phi(\lambda\alpha{:}\kappa_1.\,\kappa_2), \phi(\Pi\alpha{:}\kappa_1.\,u), \Theta, \emptyset, \mathcal{U})$ is invariant, given Θ, ϕ, and $\mathcal{U}$. This implies that if $\Theta \vdash \tau : \phi(\kappa_1)$ and τ belongs to the appropriate candidate, then we must have $(\phi(\lambda\alpha{:}\kappa_1.\,\kappa_2)\ \tau, [\tau/\alpha]\phi(u), \Theta, \emptyset, \mathcal{U})$ is invariant. By proposition C.2.69 we must prove that

    $$([\tau/\alpha]\phi(\kappa_2), [\tau/\alpha]\phi(u), \Theta, \emptyset, \mathcal{U})$$

    is invariant. But $(\phi, \alpha : \tau)$ is an instantiation that is adapted to $(\Delta, \alpha : \kappa_1)$. Applying the induction hypothesis now leads to the result.

schema formation rules This paragraph deals with rules of the form $\Delta \vdash u : \mathsf{Kscm}$.

- $u = \mathsf{Kind}$ follows directly.
- $u = z$ follows since the instantiation is adapted.
- $u = \Pi j : u_1.\,u_2$ Given Θ, ϕ, and $\mathcal{U}$ we have to prove that $(\phi(\Pi j : u_1.\,u_2), \mathsf{Kscm}, \Theta, \emptyset, \mathcal{U})$ is invariant. By the induction hypothesis, we know that $(\phi(u_1), \mathsf{Kscm}, \Theta, \emptyset, \mathcal{U})$ is invariant. The induction hypothesis also says that $([\phi, j : \kappa](u_2), \mathsf{Kscm}, \Theta, \emptyset, \mathcal{U})$ is invariant. We also know that $\Delta \vdash \kappa : \phi(u_1)$ and $(\kappa, \phi(u_1), \Theta, \emptyset, \mathcal{U})$ is invariant since the instantiation is adapted. This implies that $(\phi([\kappa/j]u_2), \mathsf{Kscm}, \Theta, \emptyset, \mathcal{U})$ is invariant.
- $u = \Pi\alpha{:}\kappa_1.\,u_1$ the proof is very similar to the above case.

□

Corollary C.2.79 *If τ is a well formed type, $|\tau|$ is strongly normalizing.*

Proof Since τ is well formed we have that $\Delta \vdash \tau : \kappa$. We only need to construct an interpretation and a mapping. For the interpretation, let $\mathcal{U}(\alpha) = \alpha$ for every type variable. Then we get $\mathcal{C}^{\mathcal{K}}_{\mathcal{U}}(\tau) = |\tau|$.

We can build the rest of $\mathcal{U}$ and $\mathcal{K}$ as:

- if $\Delta = \cdot$ then $\mathcal{U}(j) = \mathsf{Can}_0(\mathsf{Kind})$ and $\mathcal{K}(z) = \mathsf{Kind}$ for all variables j and z;
- if $\Delta = \Delta', \alpha : \kappa$ then return the $\mathcal{U}'$ and $\mathcal{K}'$ associated with Δ';
- if $\Delta = \Delta', j : u$ then $\mathcal{U} = \mathcal{U}', j : C$ and $\mathcal{K} = \mathcal{K}'$, where $C \in \mathcal{CR}(u)_{\mathcal{K}'}$ and $\mathcal{K}'$ and $\mathcal{U}'$ are associated with Δ';
- if $\Delta = \Delta', z : \mathsf{Kscm}$ then $\mathcal{K} = \mathcal{K}', z : \mathsf{Kind}$ and $\mathcal{U} = \mathcal{U}'$ where $\mathcal{K}'$ and $\mathcal{U}'$ are associated with Δ'.

□

C.2.19 Normalization of terms

In this section, we use an encoding that maps all well formed terms to types. This encoding preserves the number of reductions. The idea is similar to that of Harper *et al* [HHP93].

The encoding uses two constants. $\mathcal{A}$ is a kind and $\mathcal{B}$ is a type. $*$ is a variable that is never used, it is a wild-card.

$$\begin{array}{rl} \mathcal{A} & : \mathsf{Kind} \\ \mathcal{B} & : \Pi j : \mathsf{Kind}.\, j \\ * & \text{unused variable} \end{array}$$

The encoding for Kscm is as follows:

$$\begin{array}{rcl} S(\mathsf{Kscm}) & = & \mathsf{Kscm} \\ U(\mathsf{Kscm}) & = & \mathsf{Kind} \\ K(\mathsf{Kscm}) & = & \mathcal{A} \end{array}$$

The encoding for schemas is as follows:

$$
\begin{aligned}
U(\mathsf{Kind}) &= \mathsf{Kind} \\
U(\Pi\alpha:\kappa.\,u) &= \Pi\alpha:K(\kappa).\,U(u) \\
U(\Pi j:u_1.\,u_2) &= \Pi j:U(u_1).\,\Pi\alpha_j:K(u_1).\,U(u_2) \\
U(z) &= z
\end{aligned}
$$

$$
\begin{aligned}
K(\mathsf{Kind}) &= \mathcal{A} \\
K(\Pi\alpha:\kappa.\,u) &= \Pi\alpha:K(\kappa).\,K(u) \\
K(\Pi j:u_1.\,u_2) &= \Pi j:U(u_1).\,\Pi\alpha_j:K(u_1).\,K(u_2) \\
K(z) &= j_z
\end{aligned}
$$

$$
\begin{aligned}
T(\mathsf{Kind}) &= \mathcal{B}\ \mathcal{A} \\
T(\Pi\alpha:\kappa.\,u) &= \mathcal{B}[\mathcal{A}\to\Pi\alpha:K(\kappa).\,\mathcal{A}\to\mathcal{A}] \\
&\qquad T(\kappa)(\lambda\alpha:K(\kappa).\,T(u)) \\
T(\Pi j:u_1.\,u_2) &= \mathcal{B}[\mathcal{A}\to\Pi j:U(u_1).\,\Pi\alpha_j:K(u_1).\,\mathcal{A}\to\mathcal{A}] \\
&\qquad T(u_1)(\lambda j:U(u_1).\,\lambda\alpha_j:K(u_1).\,T(u_2)) \\
T(z) &= \alpha_z
\end{aligned}
$$

The encoding for kinds is as follows:

$$
\begin{aligned}
K(j) &= j \\
K(\Pi\alpha:\kappa_1.\,\kappa_2) &= \Pi\alpha:K(\kappa_1).\,K(\kappa_2) \\
K(\Pi j:u.\,\kappa) &= \Pi j:U(u).\,\Pi\alpha_j:K(u).\,K(\kappa) \\
K(\Pi z:\mathsf{Kscm}.\,\kappa) &= \Pi z:\mathsf{Kscm}.\,\Pi j_z:\mathsf{Kind}.\,\Pi\alpha_z:\mathcal{A}.\,K(\kappa) \\
K(\lambda j:u.\,\kappa) &= \lambda j:U(u).\,\lambda\alpha_j:K(u).\,K(\kappa) \\
K(\lambda\alpha:\kappa_1.\,\kappa_2) &= \lambda\alpha:K(\kappa_1).\,K(\kappa_2) \\
K(\kappa\ \tau) &= K(\kappa)\ T(\tau) \\
K(\kappa_1\ \kappa_2) &= K(\kappa_1)\ K(\kappa_2)T(\kappa_2) \\
K(\mathsf{Ind}(j:\mathsf{Kind})\{\vec{\kappa}\}) &= \mathsf{Ind}(j:\mathsf{Kind})\{\overrightarrow{K(\kappa)}\} \\
K(\mathsf{Elim}[\kappa,u](\tau)\{\vec{\kappa'}\}) &= \mathsf{Elim}[K(\kappa),U(u)](T(\tau))\{\overrightarrow{K(\kappa')}\}
\end{aligned}
$$

$$
\begin{aligned}
T(j) &= \alpha_j \\
T(\Pi\alpha{:}\kappa_1.\,\kappa_2) &= \mathcal{B}[\mathcal{A} \to \Pi\alpha{:}K(\kappa_1).\,\mathcal{A} \to \mathcal{A}] \\
&\qquad T(\kappa_1)(\lambda\alpha{:}K(\kappa_1).\,T(\kappa_2)) \\
T(\Pi j{:}u.\,\kappa) &= \mathcal{B}[\mathcal{A} \to \Pi j{:}U(u).\,\Pi\alpha_j{:}K(u).\,\mathcal{A} \to \mathcal{A}] \\
&\qquad T(u)(\lambda j{:}U(u).\,\lambda\alpha_j{:}K(u).\,T(\kappa)) \\
T(\Pi z{:}\mathsf{Kscm}.\,\kappa) &= \\
&\quad \mathcal{B}[\Pi z{:}\mathsf{Kscm}.\,\Pi j_z{:}\mathsf{Kind}.\,\Pi\alpha_z{:}\mathcal{A}.\,\mathcal{A} \to \mathcal{A}] \\
&\qquad (\lambda z{:}\mathsf{Kscm}.\,\lambda j_z{:}\mathsf{Kind}.\,\lambda\alpha_z{:}\mathcal{A}.\,T(\kappa)) \\
T(\lambda j{:}u.\,\kappa) &= \\
&\quad \lambda j{:}U(u).\,\lambda\alpha_j{:}K(u).\,(\lambda *{:}\mathcal{A}.\,T(\kappa))T(u) \\
T(\lambda\alpha{:}\kappa_1.\,\kappa_2) &= \lambda\alpha{:}K(\kappa_1).\,(\lambda *{:}\mathcal{A}.\,T(\kappa_2))T(\kappa_1) \\
T(\kappa\ \tau) &= T(\kappa)\ T(\tau) \\
T(\kappa_1\ \kappa_2) &= T(\kappa_1)[K(\kappa_2)]T(\kappa_2) \\
T(\mathsf{Ind}(j{:}\mathsf{Kind})\{\vec{\kappa}\}) &= \\
&\quad \mathcal{B}[(\mathsf{Kind} \to \mathcal{A} \to (\mathcal{A} \to \ldots \to \mathcal{A}) \to \mathcal{A}) \to \mathcal{A}] \\
&\quad (\lambda j{:}\mathsf{Kind}.\,\lambda\alpha_j{:}\mathcal{A}.\,\lambda Y{:}(\mathcal{A} \to \ldots \to \mathcal{A}).\,(Y\ \overrightarrow{T(\kappa_i)})) \\
T(\mathsf{Elim}[\kappa, u](\tau)\{\vec{\kappa'}\}) &= \\
&\quad \mathsf{Elim}[K(\kappa), (\lambda *{:}K(\kappa).\,K(u))](T(\tau)) \\
&\qquad \{\overrightarrow{(\lambda *{:}\mathcal{A}.\,\lambda *{:}\mathcal{A}.\,T(\kappa'_i))T(\kappa)T(u)}\}
\end{aligned}
$$

The encoding for types is as follows:

$$
\begin{aligned}
T(\alpha) &= \alpha \\
T(\lambda\alpha{:}\kappa.\,\tau) &= \lambda\alpha{:}K(\kappa).\,(\lambda *{:}\mathcal{A}.\,T(\tau))T(\kappa) \\
T(\tau_1\,\tau_2) &= T(\tau_1)\,T(\tau_2) \\
T(\lambda j{:}u.\,\tau) &= \lambda j{:}U(u).\,\lambda\alpha_j{:}K(u).\,(\lambda *{:}\mathcal{A}.\,T(\tau))T(u) \\
T(\tau[\kappa]) &= T(\tau)[K(\kappa)]T(\kappa) \\
T(\lambda z{:}\mathsf{Kscm}.\,\tau) &= \lambda z{:}\mathsf{Kscm}.\,\lambda j_z{:}\mathsf{Kind}.\,\lambda\alpha_z{:}\mathcal{A}.\,T(\tau) \\
T(\tau[u]) &= T(\tau)[U(u)][K(u)]\,T(u) \\
T(\mathsf{Ctor}\,(i,\kappa)) &= (\lambda *{:}\mathcal{A}.\,\mathsf{Ctor}\,(i,K(\kappa)))T(\kappa) \\
T(\mathsf{Elim}[\kappa,\kappa_1](\tau)\{\vec{\tau}\}) &= \\
\mathsf{Elim}[K(\kappa),K(\kappa_1)](T(\tau))\{&\overrightarrow{(\lambda *{:}\mathcal{A}.\,\lambda *{:}\mathcal{A}.\,T(\tau_i))T(\kappa)T(\kappa_1)}\}
\end{aligned}
$$

We have to define a similar transformation on contexts:

$$
\begin{aligned}
\Gamma(\cdot) &= \cdot, \mathcal{A}{:}\mathsf{Kind}, \mathcal{B}{:}\Pi j{:}\mathsf{Kind}.\,j \\
\Gamma(\Delta, \alpha{:}\kappa) &= \Gamma(\Delta), \alpha{:}K(\kappa) \\
\Gamma(\Delta, j{:}u) &= \Gamma(\Delta), j{:}U(u), \alpha_j{:}K(u) \\
\Gamma(\Delta, z{:}\mathsf{Kscm}) &= \Gamma(\Delta), z{:}\mathsf{Kscm}, j_z{:}\mathsf{Kind}, \alpha_z{:}\mathcal{A}
\end{aligned}
$$

C.2.20 Coding and reduction

In this section we state lemmas that prove that the coding preserves the number of reductions. We omit the proofs since they follow by a straightforward induction over the structure of terms.

Lemma C.2.80 *For all well typed terms A, if $A \rhd_\beta A'$, then we have*

$$
\begin{aligned}
T(A) &\rhd_\beta^{1+} T(A') \\
K(A) &\rhd_\beta^{*} K(A') \\
U(A) &\rhd_\beta^{*} U(A')
\end{aligned}
$$

Moreover, if $\|A\| \rhd_\beta A_1$, then there exists A_2 such that $\|A_2\| = A_1$ and $|T(A)| \rhd^{1+} |T(A_2)|$.

Lemma C.2.81 *For all well typed terms A, if $A \rhd_\iota A'$, then we have*

$$T(A) \rhd_\iota^{1+} T(A')$$
$$K(A) \rhd_\iota^{*} K(A')$$
$$U(A) \rhd_\iota^{*} U(A')$$

Moreover, if $\| A \| \rhd_{\iota_0} A_1$, then there exists A_2 such that $\| A_2 \| = A_1$ and $|T(A)| \rhd^{1+} |T(A_2)|$.

Lemma C.2.82 *For all well typed terms A, if $A \rhd_\eta A'$, then we have*

$$T(A) \rhd_{\beta\eta}^{1+} T(A')$$
$$K(A) \rhd_{\beta\eta}^{*} K(A')$$
$$U(A) \rhd_{\beta\eta}^{*} U(A')$$

C.2.21 Coding and typing

In this section we show that the coding of a well typed term is also well typed. For this we need to prove that the coding preserves $\beta\eta\iota$ equality. This requires a confluent calculus. Therefore, we use the unmarked terms from Section C.1.1. We extend the coding to unmarked terms by defining:

$$U(\_) = \_$$
$$K(\_) = \_$$
$$T(\_) = \_$$

It is now easy to prove by a straightforward induction on the structure of terms that the following lemma holds:

Lemma C.2.83 *Suppose $\Delta \vdash A : B$ and $B \neq$ Ext. Then we have that*

$$\Gamma(\Delta) \vdash T(A) : K(B) \text{ and } \Gamma(\Delta) \vdash K(B) : \mathsf{Kind}$$
$$\Gamma(\Delta) \vdash K(A) : U(B) \text{ and } \Gamma(\Delta) \vdash U(B) : \mathsf{Kscm} \text{ if defined}$$
$$\Gamma(\Delta) \vdash U(A) : S(B) \text{ and } \Gamma(\Delta) \vdash S(B) : \mathsf{Ext} \text{ if defined}$$

Corollary C.2.84 *Suppose $\Delta \vdash A : B$ and $B \neq$ Ext. Then $|T(A)|$ is strongly normalizing.*

C.2.22 Normalization of unmarked terms

Lemma C.2.85 *For all well typed terms A, we have that $\| A \|$ is strongly normalizing for $\beta\eta\iota_0$ reduction.*

Proof Since there can not be an infinite sequence of η reductions and we can delay η reductions, we need to prove the normalization for $\beta\iota_0$ reductions only. Suppose $\| A \|$ is not normalizing and there exists a sequence $A_1 \ldots A_i \ldots$ such that $A_i \rhd_{\beta\iota_0} A_{i+1}$ and $A_0 = \| A \|$. By lemma C.2.80 and C.2.81, we get that there exists a sequence of terms $A'_1 \ldots A'_i \ldots$ such that $\| A'_i \| = A_i$ and $| T(A'_i) | \rhd^{1+}_{\beta\iota} | T(A'_{i+1}) |$ and also $| T(A) | \rhd^{1+}_{\beta\iota} | T(A'_1) |$. This implies that $| T(A) |$ is not strongly normalizing which is a contradiction. □

C.2.23 Normalization of all terms

Lemma C.2.86 *Suppose $A \rhd_{\beta\iota} B$. Then $\| T(A) \| \rhd^{1+}_{\beta\iota} \| T(B) \|$.*

Proof By induction over the derivation of $A \rhd_{\beta\iota} B$. Note that in taking a term A to $T(A)$, all the terms C that appear as annotations at lambda abstractions are duplicated with the corresponding $T(C)$. □

Lemma C.2.87 *Suppose $\Delta \vdash A : B$. Then A is strongly normalizing.*

Proof We only have to prove normalization for $\beta\iota$ reduction. By lemma C.2.86, if A is not normalizing, then $\| T(A) \|$ is also not normalizing. But by lemma C.2.83 we have that $\Gamma(\Delta) \vdash T(A) : K(B)$ which implies (lemma C.2.85) that $\| T(A) \|$ is strongly normalizing. □

Theorem C.2.88 (Strong normalization) *All well typed terms are strongly normalizing.*

Proof Follows from lemma C.2.87. □

C.3 Church-Rosser property

The proof is structured as follows:

- We first prove that a well typed term A in $\beta\iota$ normal form has the same η reductions as $\| A \|$.
- From here we know that if A and A' are in normal form, then $\| A \|$ and $\| A' \|$ are equal. We then show that the annotations in the λ-abstractions are equal.

C.3.1 Structure of normal forms

Lemma C.3.1 *All well typed $\beta\iota$ normal terms N have the following form:*

1. $\lambda X : N_1 . N_2$.
2. $\Pi X : N_1 . N_2$.
3. $s \in \{\mathsf{Kind}, \mathsf{Kscm}, \mathsf{Ext}\}$.
4. $X\ \vec{N}$.
5. $\mathsf{Ind}(X : \mathsf{Kind})\{\vec{N}\}$.
6. $\mathsf{Ctor}\,(i, N)\ \vec{N}$ *where N is of the form 5.*
7. $\mathsf{Elim}[N, N_2](N_1)\{\vec{N}\}\ \vec{N'}$ *where N is of the form 5 and N_1 is of the form 4.*

Lemma C.3.2 *Let $\Delta, X : C, \Delta' \vdash A : B$ be a judgment and A in $\beta\iota$ normal form. If X does not occur in $FV(B) \cup FV(\Delta') \cup FV(\|A\|)$, then $X \notin FV(A)$.*

Proof The proof is by induction over the size of A. We use lemma C.3.1 to enumerate the different cases.

- The case where A is a variable or a sort is immediate.
- Suppose $\Delta, X : C, \Delta' \vdash \Pi Y : N_1 . N_2 : B$. It follows directly from the induction hypothesis that X does not occur in N_1 and N_2.
- Suppose $\Delta, X : C, \Delta' \vdash \lambda Y : N_1 . N_2 : B$ and $B = \Pi Y : N_1 . A'$. We know that $\Delta, X : C, \Delta' \vdash N_1 : s$ and therefore $X \notin FV(N_1)$. Also $B \rhd^* \Pi Y : N_1' . A''$ and $\Delta, X : C, \Delta', Y : N_1 \vdash N_2 : A''$. Since $X \notin FV(A'') \cup FV(N_1)$, we can apply the induction hypothesis and therefore $X \notin FV(N_2)$.
- Suppose $\Delta, X : C, \Delta' \vdash Y\ \vec{N} : B$. This implies that $\Delta, X : C, \Delta' \vdash Y : \Pi Z : A_1 . A_2$ and $\Delta, X : C, \Delta' \vdash N_1 : A_1$. From lemma C.1.32 and C.1.17 we have that $\Delta, X : C, \Delta' \vdash Y : \Pi Z : A_3 . A_4$ where X does not occur free in A_3 and $A_3 =_{\beta\eta\iota} A_1$ and $A_4 =_{\beta\eta\iota} A_2$. From here we can show that $\Delta, X : C, \Delta' \vdash N_1 : A_3$. We can now apply the inductive hypothesis to show that $X \notin FV(N_1)$. Iterating in this way, we can show that $X \notin FV(N_i)$.

- Suppose $\Delta, X : C, \Delta' \vdash \mathsf{Ind}(Y : \mathsf{Kind})\{\vec{N}\} : B$. Follows directly from the induction hypothesis that $X \notin FV(N_i)$.

- Suppose $\Delta, X : C, \Delta' \vdash \mathsf{Ctor}\,(i, I)\ \vec{N} : B$. Follows directly from the induction hypothesis that $X \notin FV(I)$. We can then show as above that $X \notin FV(N_i)$.

- Suppose $\Delta, X : C, \Delta' \vdash \mathsf{Elim}[N, N_1](N_2)\{\vec{N}\}\ \vec{N'} : B$. Since $\Delta, X : C, \Delta' \vdash N : \mathsf{Kind}$, it follows from the induction hypothesis that $X \notin FV(N)$. Similarly, since $\Delta, X : C, \Delta' \vdash N_1 : \mathsf{Kscm}$, or $\Delta, X : C, \Delta' \vdash N_1 : N \to \mathsf{Kind}$, it follows that $X \notin FV(N_1)$. Similarly we can prove directly from the induction hypothesis that $X \notin FV(N_2) \cup FV(\vec{N})$. Finally, as above we can prove that $X \notin FV(\vec{N'})$. □

Corollary C.3.3 *Let $\Delta \vdash A : B$. If A is in normal form, then $\| A \|$ is also in normal form.*

Proof We must show that $\| A \|$ does not contain any η reductions. The interesting case is when A is of the form $\lambda X : N_1.\, N_2\ X$. We want to show that if $X \notin FV(\| N_2 \|)$, then $X \notin FV(N_2)$. Since it is well typed we know that $\Delta \vdash \lambda X : N_1.\, N_2\ X : \Pi X : N_1.\, C$. We have that $X \notin FV(\Pi X : N_1.\, C)$. From here we get that $\Delta, X : N_1 \vdash N_2 : \Pi X : N_1.\, C$. This implies that if $X \notin FV(\| N_2 \|)$, then $X \notin FV(N_2)$. □

C.3.2 Church-Rosser

Theorem C.3.4 (Church-Rosser) *Let $\Delta \vdash A : B$ and $\Delta \vdash A' : B$ be two derivable judgments. If $A =_{\beta\eta\iota} A'$, and if A and A' are in normal form, then $A = A'$.*

Proof We know that $\| A \|$ and $\| A' \|$ are in normal form. Since the unmarked terms are confluent we have that $\| A \| = \| A' \|$. The proof is by induction over the structures of A and A'.

- The case when $A = A' = s$ or $A = A' =$ a variable is immediate.

- Suppose $A = \lambda X : N_1.\, N_2$ and $A' = \lambda X : N_1'.\, N_2'$. We know that $B =_{\beta\eta\iota} \Pi X : N_1.\, A_3 =_{\beta\eta\iota} \Pi X : N_1'.\, A_3'$. This implies that $N_1 =_{\beta\eta\iota} N_1'$ which implies that both of them have the same sort. This implies that $N_1 = N_1'$. We can now apply the induction hypothesis to N_2 and N_2' to get that $N_2 = N_2'$.

- Suppose $A = \Pi X : N_1.\, N_2$ and $A' = \Pi X : N_1'.\, N_2'$. Follows directly from the induction hypothesis.

- Suppose $A = X\ \vec{N}$ and $A' = X\ \vec{N'}$. We know that in the context Δ, the variable X has the type $\Pi\vec{Y} : \vec{B}.\, A_3$. Therefore each of the N_i and N'_i have the same type. Applying the induction hypothesis to each of them leads to the required result.

- Suppose $A = \mathsf{Ind}(X : \mathsf{Kind})\{\vec{N}\}$ and $A' = \mathsf{Ind}(X : \mathsf{Kind})\{\vec{N'}\}$. By the typing rules we know that $\Delta, X : \mathsf{Kind} \vdash N_i : \mathsf{Kind}$ and $\Delta, X : \mathsf{Kind} \vdash N'_i : \mathsf{Kind}$. Applying the induction hypothesis leads to $N_i = N'_i$.

- Suppose $A = \mathsf{Ctor}\,(i, N)\ \vec{N}$ and $A' = \mathsf{Ctor}\,(i, N')\vec{N'}$. We know that both N and N' have type Kind. The induction hypothesis directly leads to $N = N'$. We can then show as above that $N_i = N'_i$.

- Suppose $A = \mathsf{Elim}[N, N_1](N_2)\{\vec{N}\}\ \vec{N}_0$ and
  $A' = \mathsf{Elim}[N', N'_1](N'_2)\{\vec{N'}\}\ \vec{N'}_0$. Since N and N' are both of type Kind, it follows that $N = N'$. From here we get that $N_2 = N'_2$. Since both N_1 and N'_1 have the type Kscm or have the type $N \to \mathsf{Kind}$, it follows that $N_1 = N'_1$. From this we can show that the N_i and N'_i are equal. Finally as above, we can show that the N_{0i} and the N'_{0i} are equal. □

Theorem C.3.5 (Confluence) *Let $\Delta \vdash A : B$ and $\Delta \vdash A' : B$ be two judgments. If $A =_{\beta\eta\iota} A'$, then A and A' have a common reduct – there exists a term C such that $A \rhd^* C$ and $A' \rhd^* C$.*

Proof We know that both A and A' reduce to normal forms A_1 and A'_1. Due to subject reduction, both A_1 and A'_1 have the same type B. By the previous lemma $A_1 = A'_1$. □

C.4 Consistency

Theorem C.4.1 (Consistency of the logic) *There exists no term A for which $\cdot \vdash A : \Pi X : \mathsf{Kind}.\, X$.*

Proof Suppose there exists a term A for which $\cdot \vdash A : \Pi X : \mathsf{Kind}.\, X$. By theorem C.2.88, there exists a normal form B for A. By the subject reduction $\cdot \vdash B : \Pi X : \mathsf{Kind}.\, X$. We can show now that this leads to a contradiction by case analysis of the possible normal forms for the types in the calculus. □

Bibliography

[AF00a] Andrew W. Appel and Edward W. Felten. Models for security policies in proof-carrying code. Technical report, Princeton University, Department of Computer Science, July 2000.

[AF00b] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. 27th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 243–253. ACM Press, 2000.

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[Asp95] David Aspinall. Subtyping with singleton types. In *Proc. 1994 CSL*. Springer Lecture Notes in Computer Science, 1995.

[Bar91] Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science (volume 2)*. Oxford University Press, 1991.

[Bar99] Henk P. Barendregt. Proof-assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers B.V., 1999.

[BB95] Corrado Bohm and Alessandro Berarducci. Automatic synthesis of typed λ-programs on term algebras. In *Theoretical Computer Science*, pages 39:135–154, 1995.

[BHS99] G. Barthe, J. Hatcliff, and M. Sorensen. CPS translations and applications: the cube and beyond. *Higher Order and Symbolic Computation*, 12(2):125–170, September 1999.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[CLN+00] Christopher Cobly, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for java. In *Proc. ACM SIGPLAN '00 Conf. on Prog. Lang. Design and Implementation*, pages 95–107, New York, 2000. ACM Press.

[CP96] Iliano Cervesato and Frank Pfenning. A linear logical framework. In *Proc. 11th IEEE Symp. on Logic in Computer Science*, pages 264–275, 1996.

[Cra00] Karl Crary. Typed assembly language: Type theory for machine code. Talk presented at 2000 PCC Workshop, Santa Barbara, CA, June 2000.

[CV01] Karl Crary and J Vanderwaart. An expressive, scalable type theory for certified code. Technical Report CMU-CS-01-113, Carnegie Mellon University, 2001.

[CW99] Karl Crary and Stephanie Weirich. Flexible type analysis. In *Proc. 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 233–248. ACM Press, September 1999.

[CW00] Karl Crary and Stephanie Weirich. Resource bound certification. In *Proc. Twenty-Seventh Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 184–198. ACM Press, 2000.

[CWM98] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 301–312. ACM Press, September 1998.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, TX, January 1999.

[De 80] N. De Bruijn. A survey of the project AUTOMATH. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Edited by J. P. Seldin and J. R. Hindley, Academic Press, 1980.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control, a study of the cps transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[DRW95] Catherine Dubois, Francois Rouaix, and Pierre Weis. Extensional polymorphism. In *Proc. Principles of Programming Languages*. ACM Press, 1995.

[Dug98] Dominic Duggan. A type-based semantics for user defined marshalling in polymorphic languages. In *Second Types in Compilation Workshop*, 1998.

[Geu93] J. Geuvers. *Logics and Type Systems*. PhD thesis, Catholic University of Nijmegen, The Netherland, 1993.

[Gim98] Eduardo Gimenez. A tutorial on recursive types in Coq, May 1998.

[Gir72] J. Y. Girard. *Interpretation Fonctionnelle et Elimination des Coupures dans l'Arithmetique d'Ordre Superieur*. PhD thesis, University of Paris VII, 1972.

[Gir88] Jean-Yves Girard. Typed lambda-calculus, draft book translated by paul taylor and yves lafont, 1988.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[Hay91] S Hayashi. Singleton, union, and intersection types for program extraction. In *Proc. International Conference on Theoretical Aspects of Computer Software*, pages 701–730, 1991.

[HHP93] Bob Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, jan 1993.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, pages 130–141, New York, Jan 1995. ACM Press.

[HM98] Robert Harper and Greg Morrisett. Typed closure conversion for recursively-defined functions. In *Second International Workshop on Higher Order Operational Techniques in Semantics (HOOTS98*, New York, Sep 1998. ACM Press.

[HM99] Robert Harper and John C. Mitchell. Parametricity and variants of Girard's J operator. *Information Processing Letters*, 70(1):1–5, April 1999.

[How80] W. A. Howard. The formulae-as-types notion of constructions. In *To H.B.Curry: Essays on Computational Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[HPM$^+$00] Gerard Huet, Christine Paulin-Mohring, et al. The Coq proof assistant reference manual. Part of the Coq system version 6.3.1, May 2000.

[Ler92] Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 177–188, New York, Jan 1992. ACM Press. Longer version available as INRIA Tech Report.

[LST99] Christopher League, Zhong Shao, and Valery Trifonov. Representing Java classes in a typed intermediate language. In *Proc. 1999 ACM SIGPLAN International Conf. on Functional Programming (ICFP'99)*, pages 183–196. ACM Press, September 1999.

[Min97] Yasuhiko Minamide. Full lifting of type parameters. Technical report, RIMS, Kyoto University, 1997.

[MMH96] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proc. 23rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 271–283. ACM Press, 1996.

[Mor95] Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, December 1995. CMU-CS-95-223.

[Mor00] Greg Morrisett. Open problems for certifying compilers. Talk presented at 2000 PCC Workshop, Santa Barbara, CA, June 2000.

[MSS00] Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. Technical Report YALEU/DCS/TR1205, Dept. of Computer Science, Yale University, New Haven, CT, November 2000. Available at `http://flint.cs.yale.edu`.

[MSS01] Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *Proc. 2001 ACM SIGPLAN Programming Language Design and Implementation (PLDI'01)*. ACM Press, June 2001.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *Proc. 25rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, page (to appear). ACM Press, 1998.

[Nad94] Gopalan Nadathur. A notation for lambda terms II: Refinements and applications. Technical Report CS-1994-01, Duke University, Durham, NC, January 1994.

[Nec97] George Necula. Proof-carrying code. In *Twenty-Fourth Annual ACM Symp. on Principles of Prog. Languages*, New York, Jan 1997. ACM Press.

[Nec98] George Ciprian Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, September 1998.

[NL96] George Necula and Peter Lee. Safe kernel extensions without runtime checking. In *2nd USENIX Symposium on Operating System Design and Implementation*, pages 229–243, 1996.

[NL98] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proc. ACM SIGPLAN '98 Conf. on Prog. Lang. Design and Implementation*, pages 333–344, New York, 1998. ACM Press.

[NW90] Gopalan Nadathur and Debra Sue Wilson. A representation of lambda terms suitable for operations on their intensions. In *1990 ACM Conference on Lisp and Functional Programming*, pages 341–348, New York, June 1990. ACM Press.

[Pau89] C. Paulin-Mohring. Extracting f_ω's programs from proofs in the calculus of constructions. In *Sixteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 89–104, New York, Jan 1989. ACM Press.

[Pau93] C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In M. Bezem and J. Groote, editors, *Proc. TLCA*. LNCS 664, Springer-Verlag, 1993.

[PDM89] Benjamin Pierce, Scott Dietzen, and Spiro Michaylov. Programming in higher-order typed lambda-calculi. Technical Report CMU-CS-89-111, Carnegie Mellon University, 1989.

[Pey92] Simon Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[Pfe88] Frank Pfenning. Inductively defined types in the calculus of constructions. Ergo report 88-069, Dept. of Computer Science, Carnegie Mellon University, November 1988.

[PJ93] John Peterson and Mark P. Jones. Implementing type classes. In *Programming Language Design and Implementation*, New York, 1993. ACM Press.

[PL89] Frank Pfenning and Peter Lee. Leap: a language with eval and polymorphism. In *International Joint Conference on Theory and Practice of Software Development*, 1989.

[Rep91] John H. Reppy. CML: A higher-order concurrent language. In *Proc. ACM SIGPLAN '91 Conf. on Prog. Lang. Design and Implementation*, pages 293–305. ACM Press, 1991.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Proceedings, Colloque sur la Programmation, Lecture Notes in Computer Science, volume 19*, pages 408–425. Springer-Verlag, Berlin, 1974.

[SA95] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proc. ACM SIGPLAN '95 Conf. on Prog. Lang. Design and Implementation*, pages 116–129. ACM Press, 1995.

[SG90] Mark A. Sheldon and David K. Gifford. Static dependent types for first class modules. In *1990 ACM Conference on Lisp and Functional Programming*, pages 20–29, New York, June 1990. ACM Press.

[Sha97a] Zhong Shao. Flexible representation analysis. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 85–98. ACM Press, June 1997.

[Sha97b] Zhong Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[Sha98] Zhong Shao. Typed cross-module compilation. In *Proc. 1998 ACM SIGPLAN International Conf. on Functional Programming*. ACM Press, 1998.

[Sha99] Zhong Shao. Transparent modules with fully syntactic signatures. In *Proc. 1999 ACM SIGPLAN International Conf. on Functional Programming (ICFP'99)*, pages 220–232. ACM Press, September 1999.

[Sha01] Zhong Shao. Yale University, personal communication, 2001.

[SLM98] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *Proc. International Conference of Functional Programming*. ACM Press, 1998.

[SSTP01] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. Technical Report YALEU/DCS/TR1211, Dept. of Computer Science, Yale University, New Haven, CT, July 2001. Available at `http://flint.cs.yale.edu`.

[SSTP02] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. In *Proc. 2002 ACM SIGPLAN/SIGACT Principles of Programming Languages (POPL'02)*. ACM Press, January 2002.

[STS00a] Bratin Saha, Valery Trifonov, and Zhong Shao. Fully reflexive intensional type analsysis in a type erasure semantics. In *Proc. 2000 Types in Compilation Workshop*, September 2000.

[STS00b] Bratin Saha, Valery Trifonov, and Zhong Shao. Intensional analysis of quantified types. Technical Report YALEU/DCS/TR-1194, Dept. of Computer Science, Yale University, New Haven, CT, March 2000.

[Tar96] David Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML*. PhD thesis, Carnegie Mellon University, December 1996. CMU-CS-97-108.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conf. on Prog. Lang. Design and Implementation*, pages 181–192. ACM Press, 1996.

[TO98] Andrew Tolmach and Dino P. Oliva. From ml to ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

[Tof98] Mads Tofte. Region-based memory management (invited talk). In *Proc. 1998 Types in Compilation Workshop*, March 1998.

[TSS00] Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analsysis. In *Proc. 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 82–93. ACM Press, September 2000.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 188–201. ACM Press, 1994.

[WA99] Daniel C. Wang and Andrew W. Appel. Safe garbage collection = regions + intensional type analysis. Technical Report TR-609-99, Princeton University, 1999.

[WA01] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In *Proc. 28th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, page (to appear). ACM Press, 2001.

[WC94] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. In *1994 ACM Conference on Lisp and Functional Programming*, pages 250–262, New York, June 1994. ACM Press.

[Wer94] Benjamin Werner. *Une Theorie des Constructions Inductives*. PhD thesis, University of Paris, 1994.

[WF92] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical report, Dept. of Computer Science, Rice University, June 1992.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proc. Twenty-Sixth ACM Symp. on Principles of Programming Languages*, pages 214–227. ACM Press, 1999.

[Yan98] Zhe Yang. Encoding types in ML-like languages. In *Proc. 1998 ACM SIGPLAN International Conf. on Functional Programming*, pages 289–300. ACM Press, 1998.